	Type	L #	Hits	Search Text	DBs	Time Stamp
1	BRS	L1	0	710/262.ccls	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/01/20 15:21
2	BRS	L2	210	710/262.ccls.	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/01/20 15:21
3	BRS	L3	3063	interrupt near2 mask$3	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/01/20 15:37
4	BRS	L4	3176	2 or 3	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/01/20 15:21
5	BRS	L5	324	(interrupt near2 mask$3).ti,ab.	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/01/20 15:22
6	BRS	L6	509	2 or 5	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/01/20 15:22

	Type	L #	Hits	S arch Text	DBs	Time Stamp
7	BRS	L7	2647	(realtime or real adj time) adj (OS or operating adj system)	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/01/20 15:23
8	BRS	L8	123	multitask$3 adj OS	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/01/20 15:23
9	BRS	L9	2762	7 or 8	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/01/20 15:23
10	BRS	L10	19	6 and 9	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/01/20 15:23
11	BRS	L11	16	interrupt near2 mask$3 near4 cancel$6	USPAT; US-PGPUB; EPO; JPO; DERWENT; IBM_TDB	2004/01/20 15:38

US005768599A

United States Patent [19]

Yokomizo

[11] Patent Number: 5,768,599

[45] Date of Patent: Jun. 16, 1998

[54] **INTERRUPT MANAGING SYSTEM FOR REAL-TIME OPERATING SYSTEM**

[75] Inventor: **Takashi Yokomizo**, Tokyo, Japan

[73] Assignee: **NEC Corporation**, Tokyo, Japan

[21] Appl. No.: **607,461**

[22] Filed: **Feb. 27, 1996**

[30] **Foreign Application Priority Data**

Feb. 28, 1995 [JP] Japan 7-040228

[51] **Int. Cl.**[6] **G06F 13/24**

[52] **U.S. Cl.** **395/733**; 395/734; 395/735; 395/738; 395/868; 395/869

[58] **Field of Search** 395/733, 734, 395/735, 868, 869, 737, 738–742

[56] **References Cited**

U.S. PATENT DOCUMENTS

5,214,650	5/1993	Renner et al.	370/276
5,448,743	9/1995	Gulick et al.	395/869
5,566,334	10/1996	Loader	395/733
5,619,706	4/1997	Young	395/733

FOREIGN PATENT DOCUMENTS

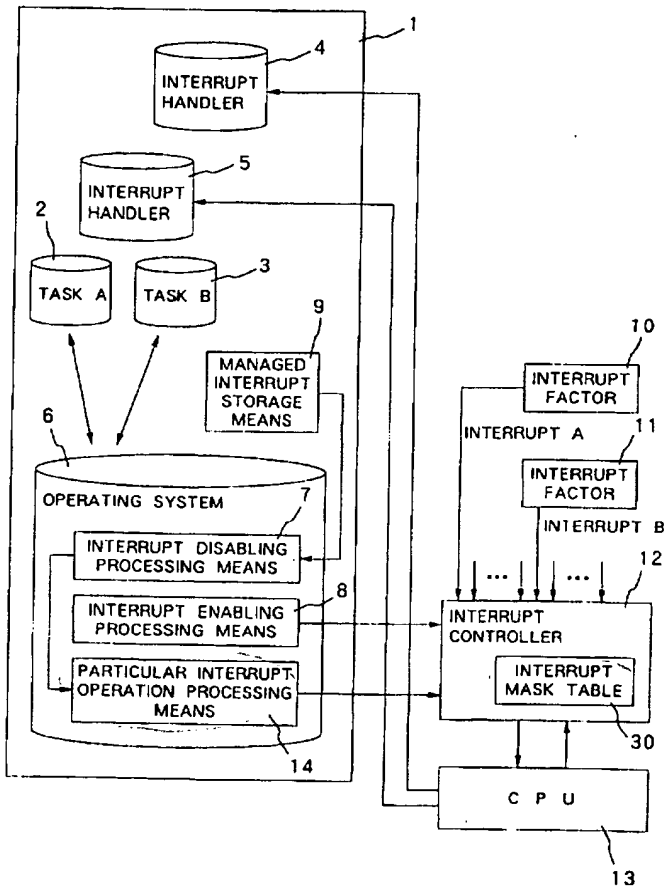
2-220138 9/1990 Japan .

Primary Examiner—Jack B. Harvey
Assistant Examiner—Ario Etienne
Attorney, Agent, or Firm—Foley & Lardner

[57] **ABSTRACT**

An interrupt managing system for a computer system in which resources are managed by a real-time operating system. The interrupt managing system has managed interrupt storage unit in which information regarding interrupts to be managed by the real-time operating system is stored, interrupt disabling processing unit for reading out the information stored in the managed interrupt storage unit and disabling an interrupt designated by the information in order to perform exclusive control needed for system call processing, and interrupt enabling processing unit for enabling the disabled interrupt. According to the interrupt managing system, an asynchronous interrupt which does not have an influence on the resource management by an OS is enabled even during a system call processing and an interrupt which does not issue a system call can be processed without any delay.

10 Claims, 11 Drawing Sheets



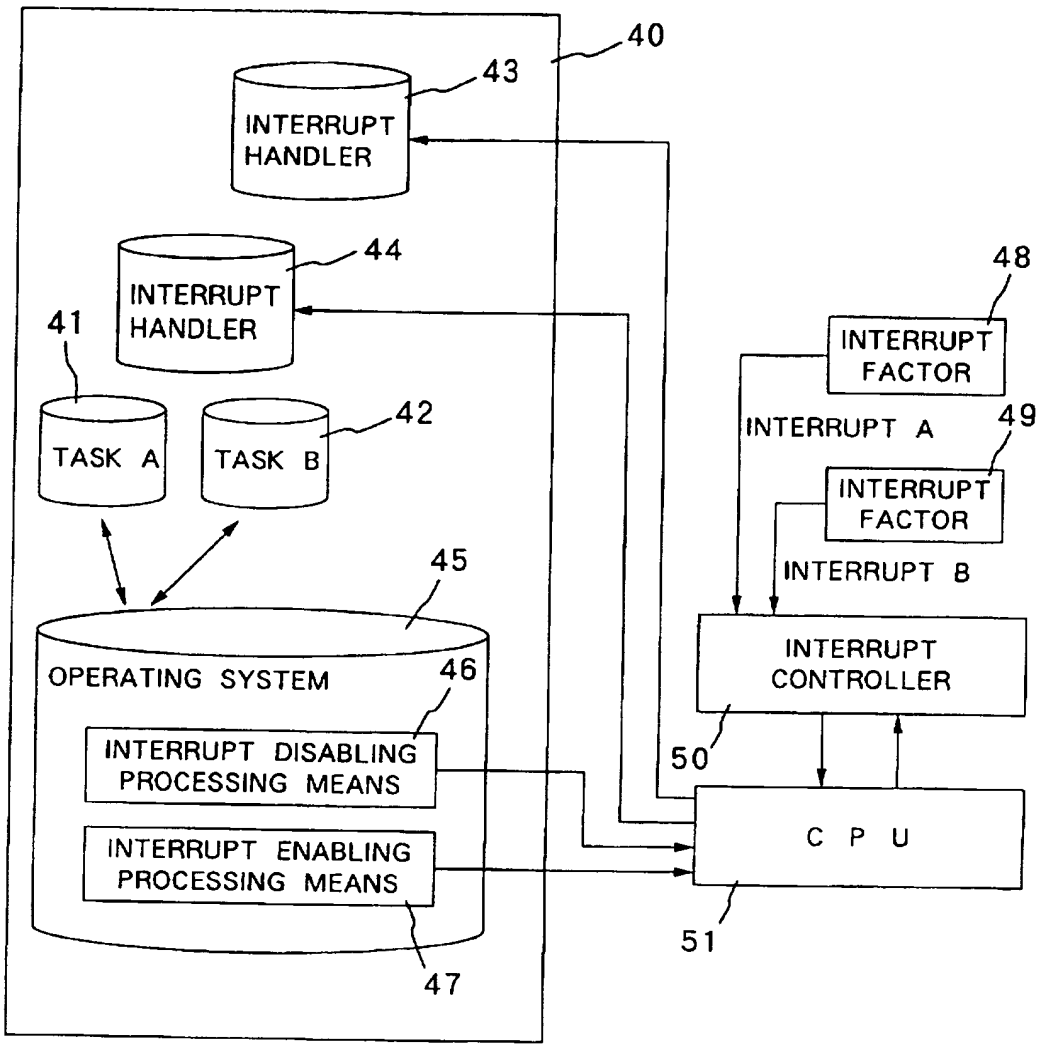


FIG. 1 (PRIOR ART)

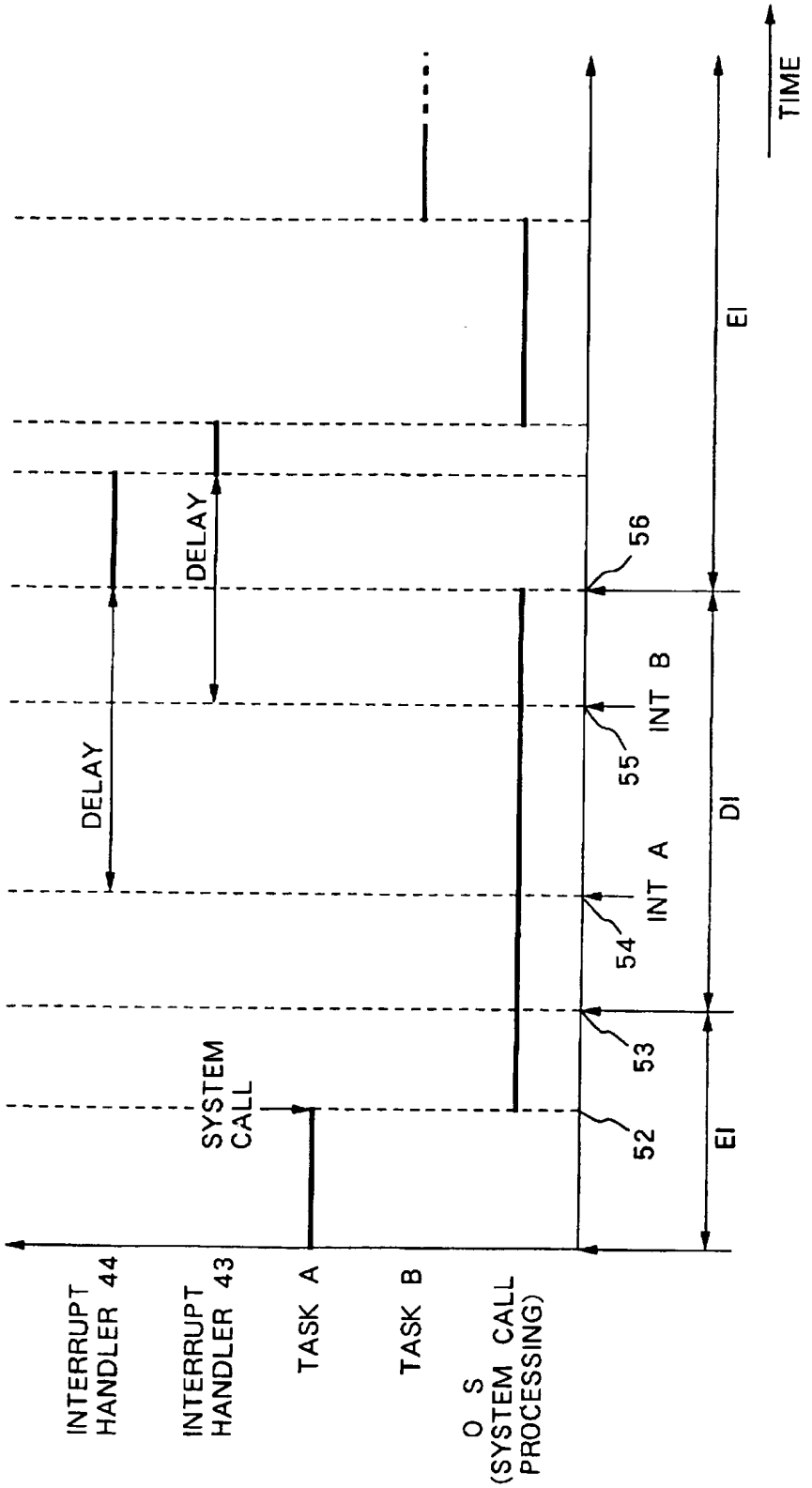


FIG. 2 (PRIOR ART)

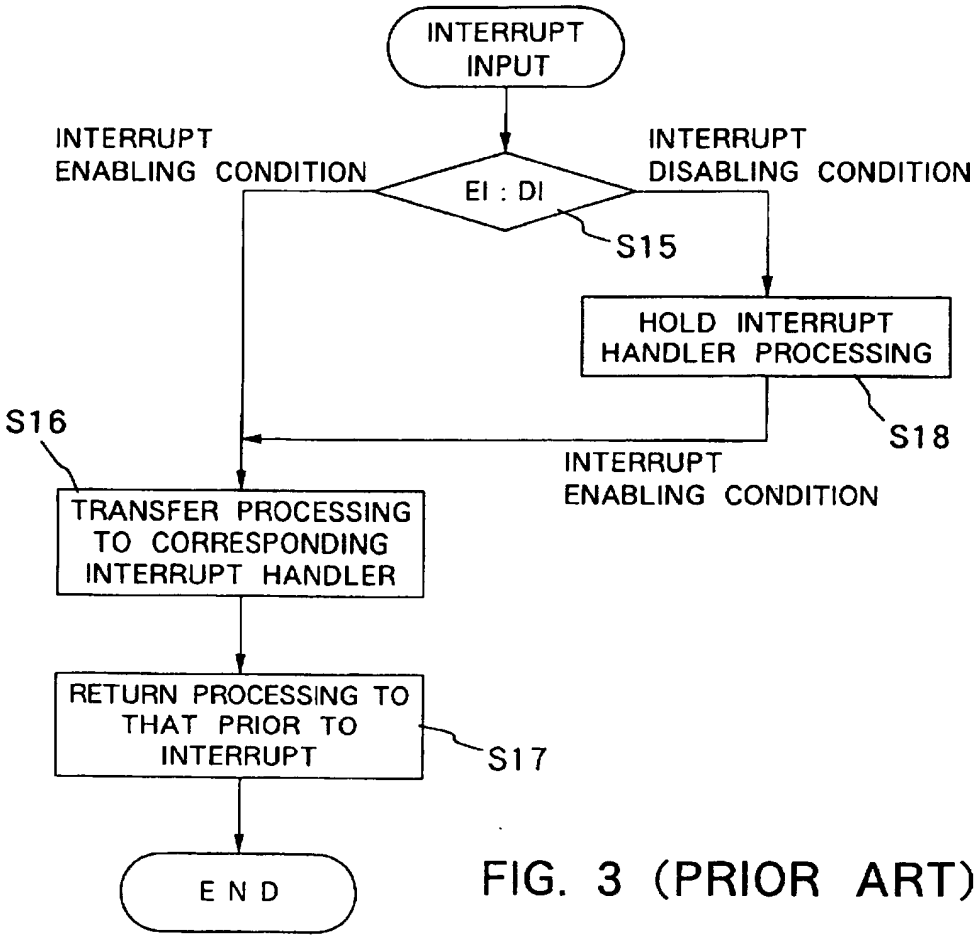


FIG. 3 (PRIOR ART)

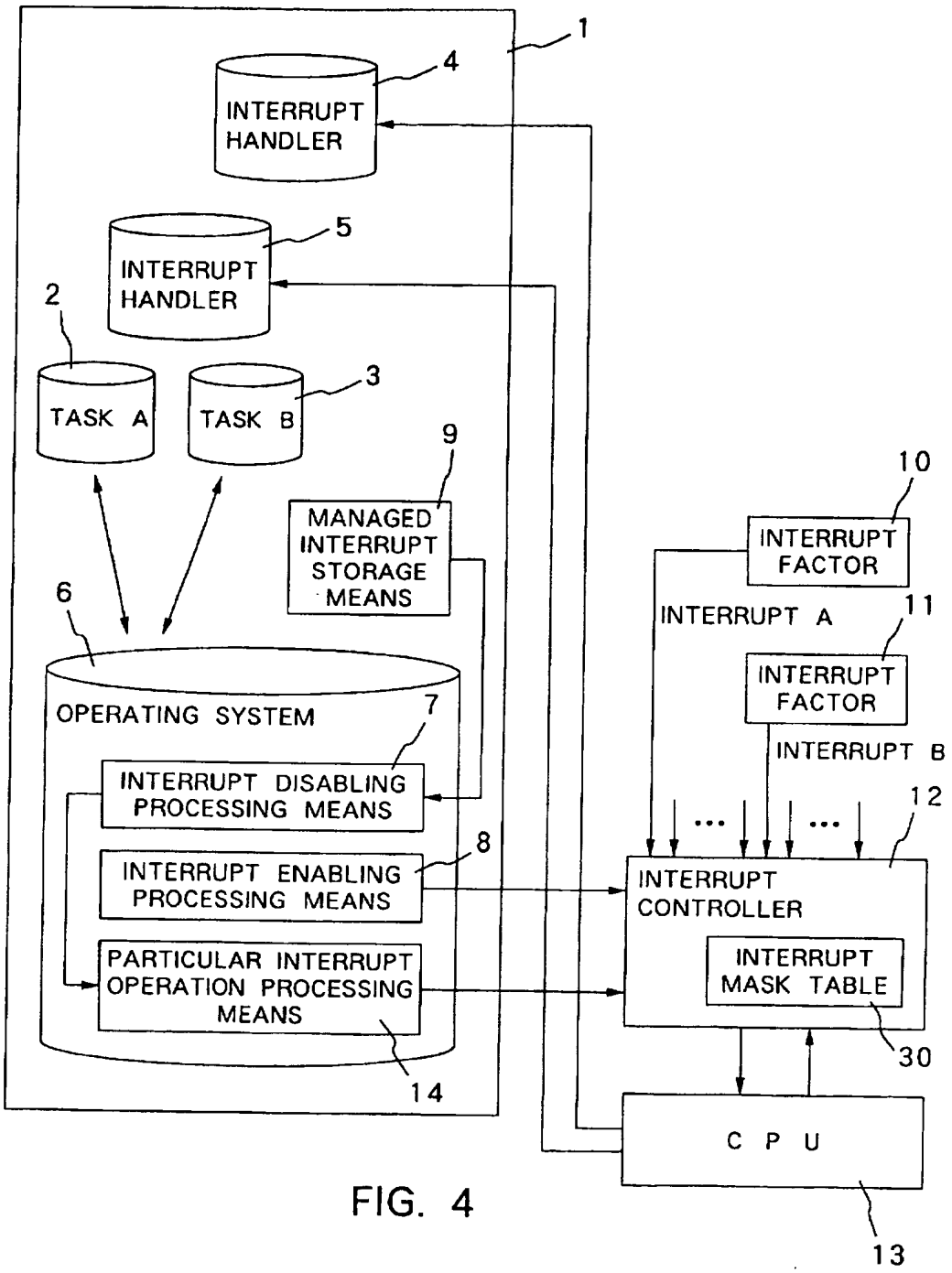

1
4
INTERRUPT HANDLER
5
INTERRUPT HANDLER
2
TASK A
TASK B
3
9
MANAGED INTERRUPT STORAGE MEANS
6
OPERATING SYSTEM
7
INTERRUPT DISABLING PROCESSING MEANS
8
INTERRUPT ENABLING PROCESSING MEANS
PARTICULAR INTERRUPT OPERATION PROCESSING MEANS
14
10
INTERRUPT FACTOR
INTERRUPT A
11
INTERRUPT FACTOR
INTERRUPT B
12
INTERRUPT CONTROLLER
INTERRUPT MASK TABLE
30
C P U
13

FIG. 4

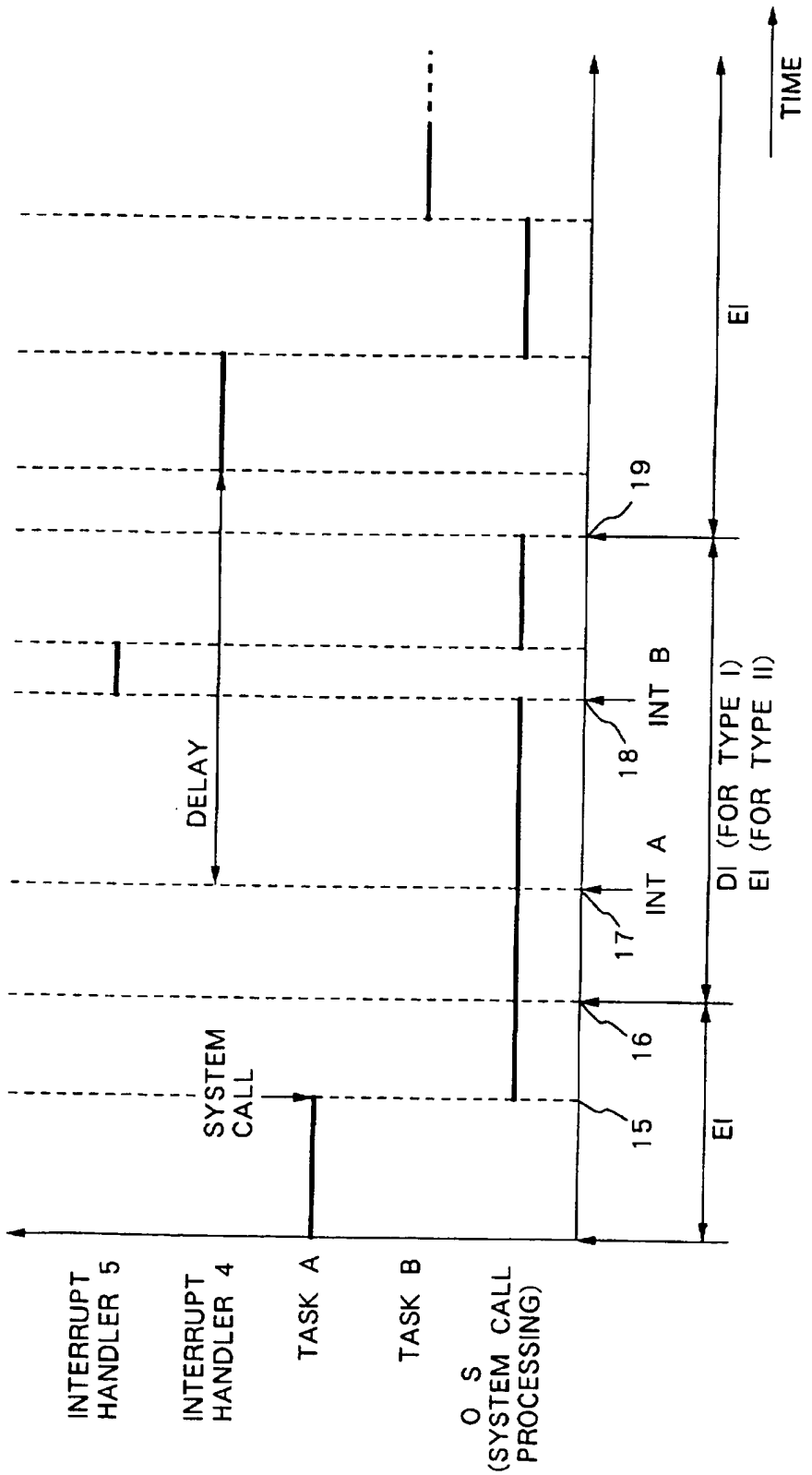

TIME
EI
19
INT B
18
DI (FOR TYPE I)
EI (FOR TYPE II)
INT A
17
DELAY
16
SYSTEM CALL
15
EI
INTERRUPT HANDLER 5
INTERRUPT HANDLER 4
TASK A
TASK B
O S (SYSTEM CALL PROCESSING)

FIG. 5

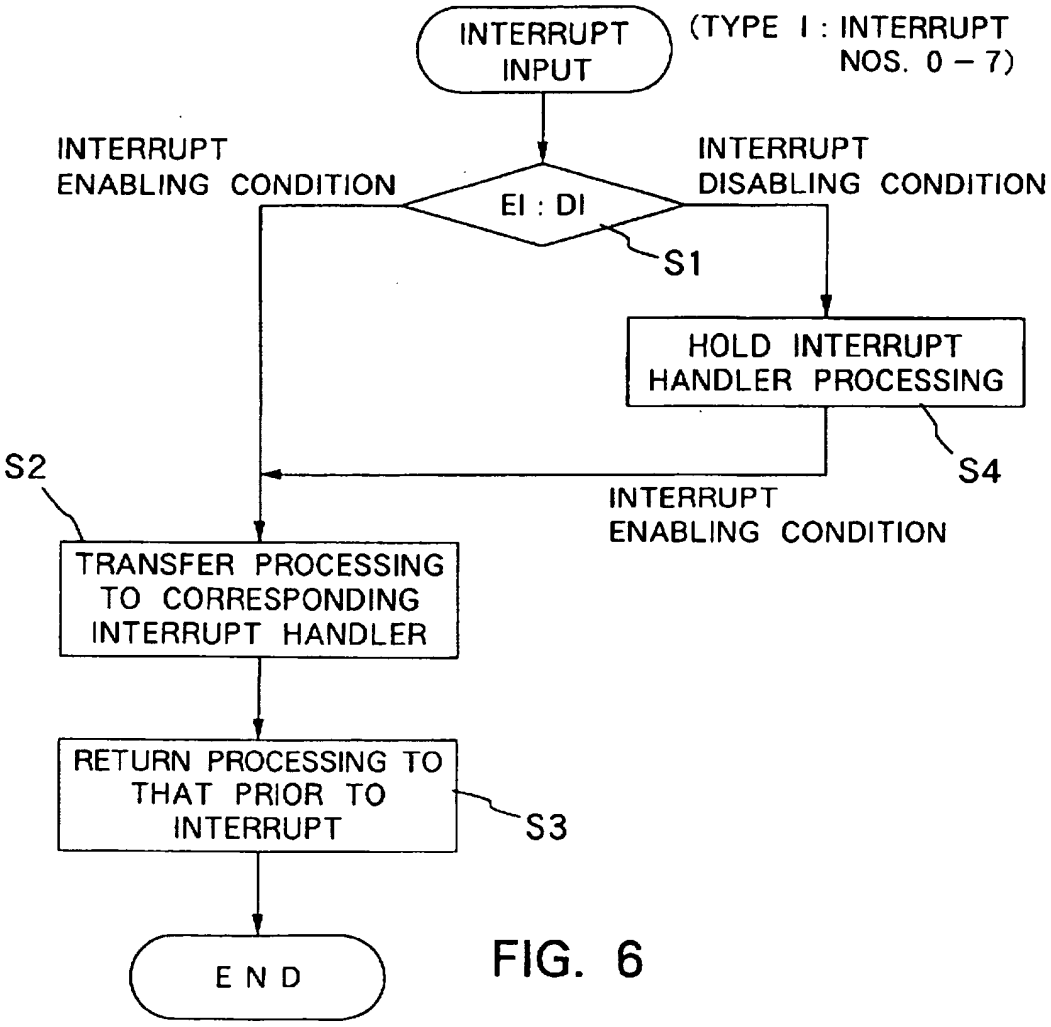

INTERRUPT INPUT
(TYPE I : INTERRUPT NOS. 0 – 7)
INTERRUPT ENABLING CONDITION
EI : DI
INTERRUPT DISABLING CONDITION
S1
HOLD INTERRUPT HANDLER PROCESSING
S4
S2
INTERRUPT ENABLING CONDITION
TRANSFER PROCESSING TO CORRESPONDING INTERRUPT HANDLER
RETURN PROCESSING TO THAT PRIOR TO INTERRUPT
S3
END

FIG. 6

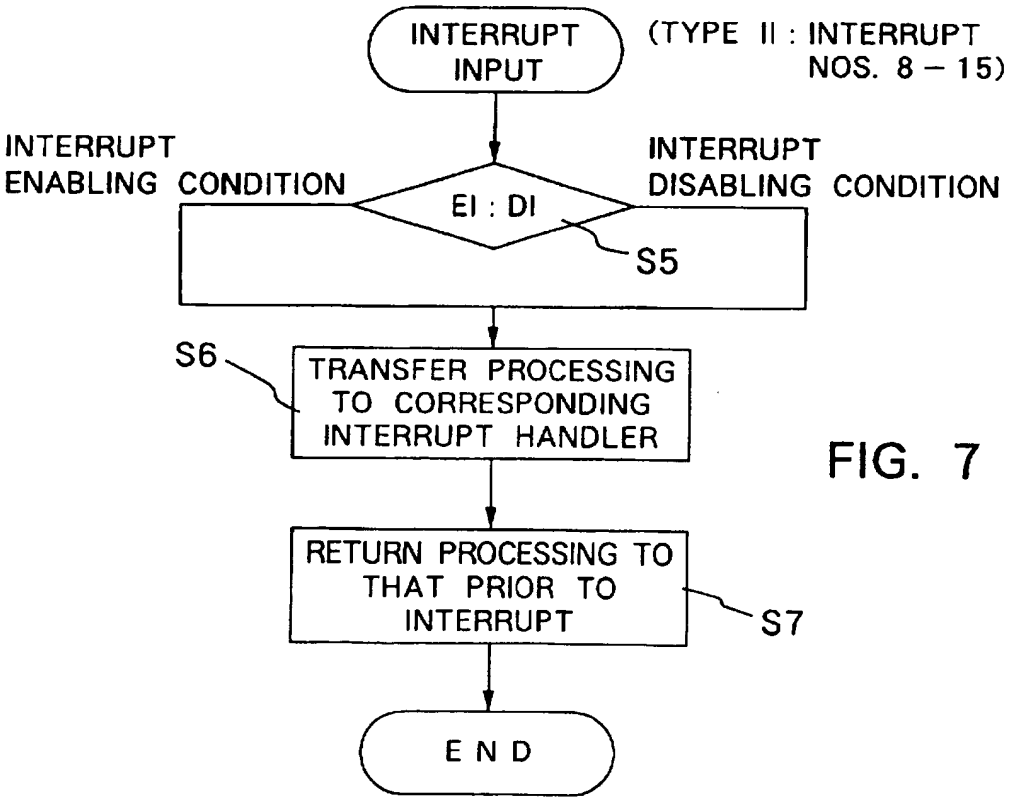

INTERRUPT INPUT
(TYPE II : INTERRUPT NOS. 8 – 15)
INTERRUPT ENABLING CONDITION
INTERRUPT DISABLING CONDITION
EI : DI
S5
S6
TRANSFER PROCESSING TO CORRESPONDING INTERRUPT HANDLER
RETURN PROCESSING TO THAT PRIOR TO INTERRUPT
S7
E N D

FIG. 7

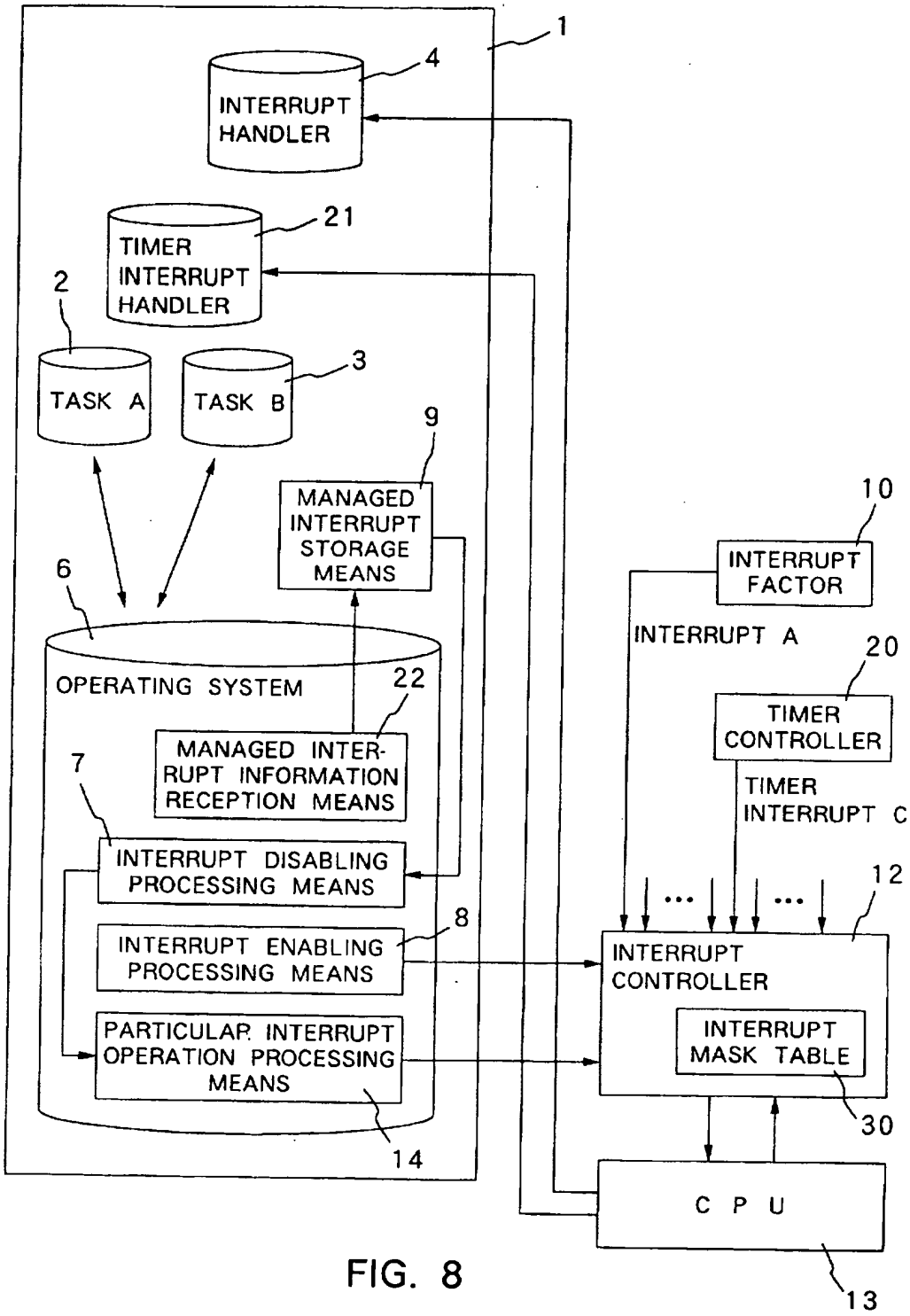

1
4
INTERRUPT HANDLER
21
TIMER INTERRUPT HANDLER
2
TASK A
TASK B
3
9
MANAGED INTERRUPT STORAGE MEANS
10
INTERRUPT FACTOR
INTERRUPT A
6
OPERATING SYSTEM
22
20
TIMER CONTROLLER
TIMER INTERRUPT C
MANAGED INTERRUPT INFORMATION RECEPTION MEANS
7
INTERRUPT DISABLING PROCESSING MEANS
12
8
INTERRUPT ENABLING PROCESSING MEANS
INTERRUPT CONTROLLER
INTERRUPT MASK TABLE
PARTICULAR INTERRUPT OPERATION PROCESSING MEANS
30
14
C P U
13

FIG. 8

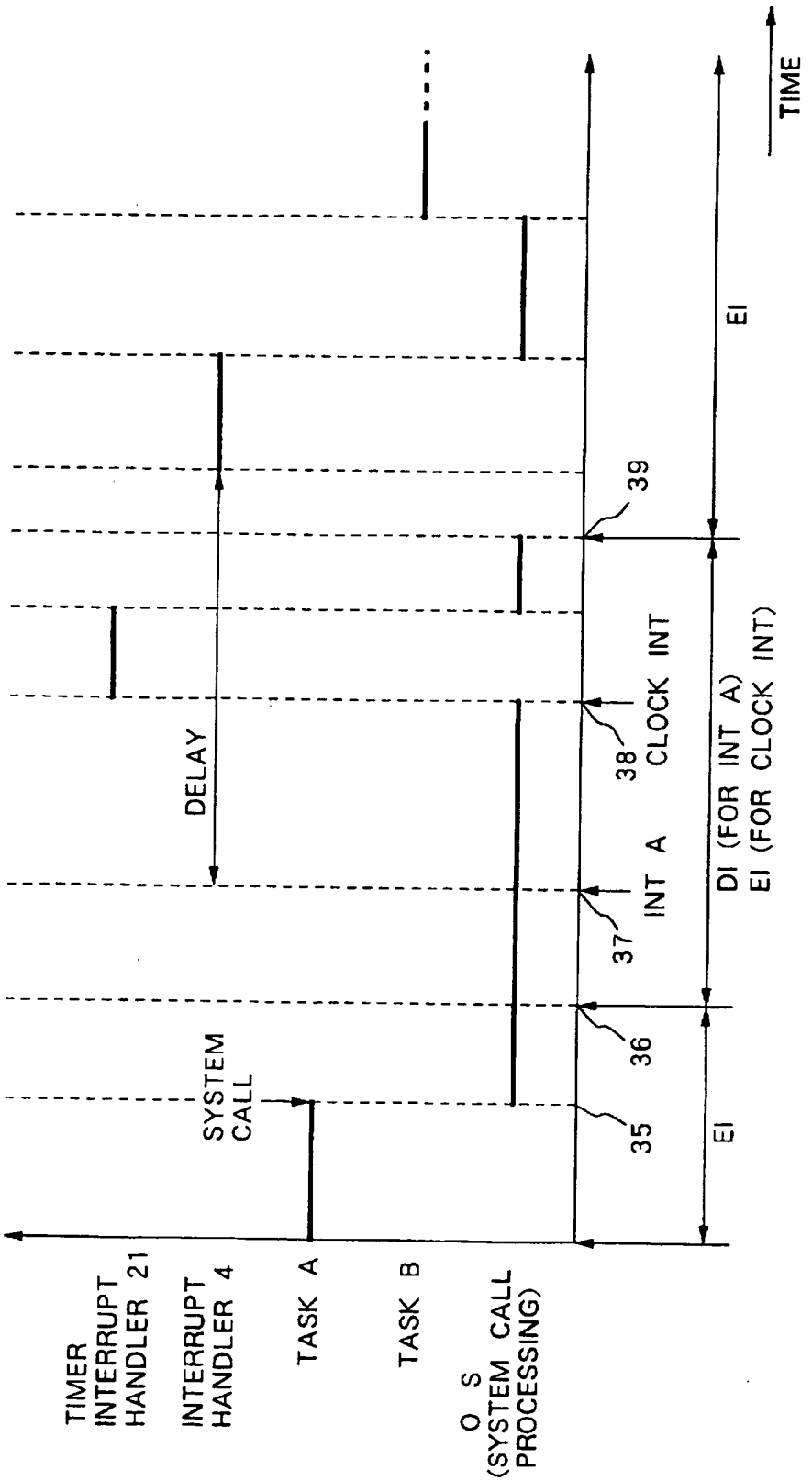

TIME
EI
39
DELAY
38
CLOCK INT
INT A
37
DI (FOR INT A)
EI (FOR CLOCK INT)
36
SYSTEM CALL
35
EI
TIMER INTERRUPT HANDLER 21
INTERRUPT HANDLER 4
TASK A
TASK B
O S (SYSTEM CALL PROCESSING)

FIG. 9

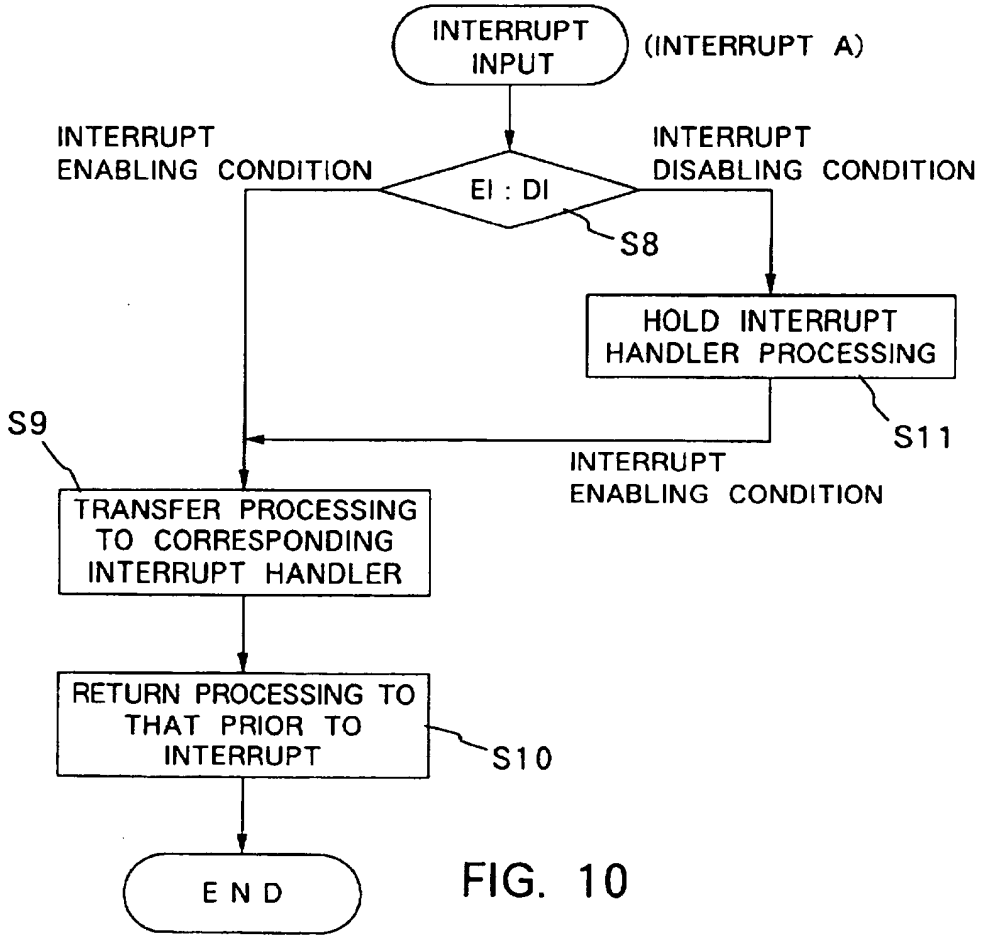

INTERRUPT INPUT
(INTERRUPT A)
INTERRUPT ENABLING CONDITION
EI : DI
INTERRUPT DISABLING CONDITION
S8
HOLD INTERRUPT HANDLER PROCESSING
S11
S9
INTERRUPT ENABLING CONDITION
TRANSFER PROCESSING TO CORRESPONDING INTERRUPT HANDLER
RETURN PROCESSING TO THAT PRIOR TO INTERRUPT
S10
END

FIG. 10

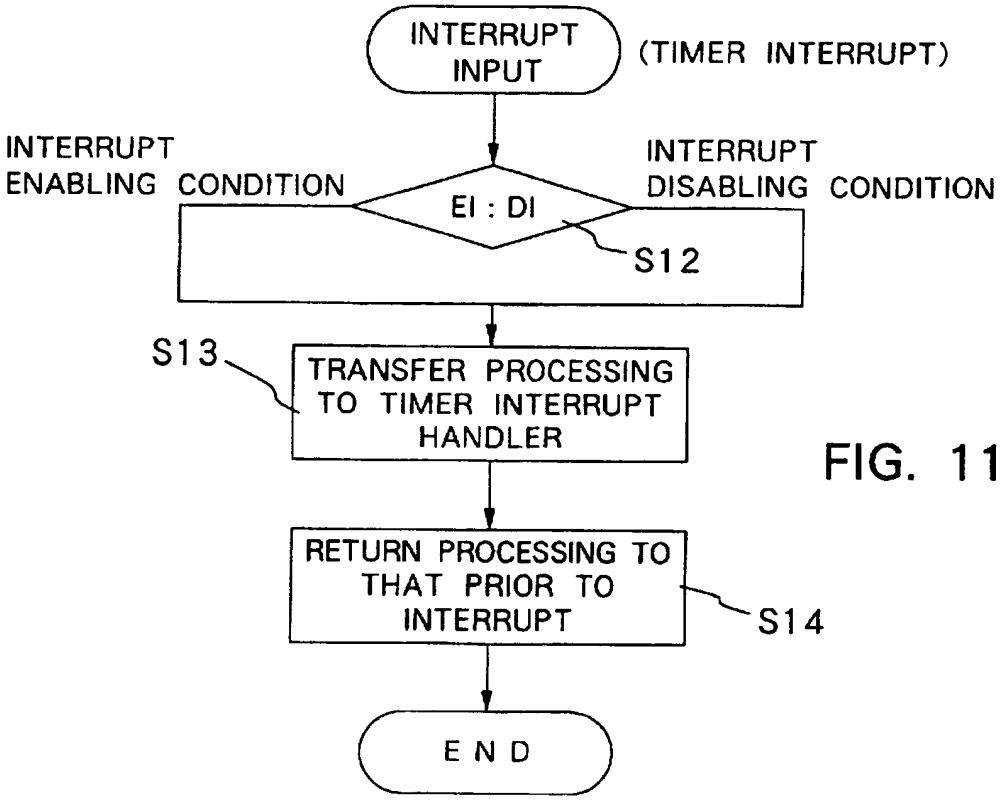

INTERRUPT INPUT
(TIMER INTERRUPT)
INTERRUPT ENABLING CONDITION
INTERRUPT DISABLING CONDITION
EI : DI
S12
S13
TRANSFER PROCESSING TO TIMER INTERRUPT HANDLER
RETURN PROCESSING TO THAT PRIOR TO INTERRUPT
S14
E N D

FIG. 11

INTERRUPT MANAGING SYSTEM FOR REAL-TIME OPERATING SYSTEM

BACKGROUND OF THE INVENTION

1. Field of the Invention

This invention relates to an interrupt managing system for a real-time operating system (real-time OS).

2. Description of the Prior Art

In a computer system wherein resources and processes are controlled by a real-time OS, also interrupt processing for an asynchronous interrupt which may be generated by various factors is managed by the real-time OS. A software routine for executing interrupt processing is normally called an interrupt handler. In a conventional computer system which is controlled by a real-time OS, in order to allow a system call to the OS to be issued from all interrupt handlers, it is common for the real-time OS to manage all interrupts collectively. The system call signifies an instruction issued by a task or an interrupt handler in order to request a service provided by the OS.

FIG. **1** shows a conventional computer system controlled by a real-time OS. To a CPU (Central Processing Unit) **51**, an interrupt controller **50** which transmits generation of an interrupt event to the CPU **51** and a main storage **40** are connected. An interrupt A from an interrupt factor **48** and another interrupt B from another interrupt factor **49** are inputted to the interrupt controller **50**. An OS **45** is resident in the main storage **40**, and a routine **41** of a task A, another routine **42** of another task B, an interrupt handler **44** which starts in accordance with the occurence of the interrupt A and another interrupt handler **43** which starts in accordance with the occurence of the interrupt B are stored in the main storage **40**. It is assumed that, among them, the interrupt handler **44** corresponding to the interrupt A executes its software routine to issue a system call to the OS **45**, but the interrupt handler **43** corresponding to the interrupt B does not issue a system call in its software routine. Further, the OS **45** includes, in the inside thereof, interrupt disabling processing means **46** for putting the CPU **51** into an interrupt disabling condition, and interrupt enabling processing means **47** for putting the CPU **51** into an interrupt enabling condition.

By the way, some system call processing routines executed by the OS in response to generation of a system call necessitate exclusive control in order to prevent such conflict of resources that the same resource is handled in response to system calls generated from different interrupt handlers. The exclusive control is realized by setting a particular interval of the system call processing routine as interrupt disabling interval and inhibiting acceptance of any interrupt by the CPU **51** during execution of the routine within the interrupt inhibition interval.

The interrupt processing of the above computer system is described below with reference to FIG. **2**. Here, description is given of the case wherein a system call to the OS **45** is issued by the task A during execution of the task A and then, within an interrupt disabling interval in the system call processing, the interrupt A is inputted from the interrupt factor **48** to the interrupt controller **50** and then the interrupt B is inputted from the interrupt factor **49** to the interrupt controller **50**. In FIG. **2**, each routine being executed by the CPU **51** is indicated by a thick line. Further, EI (enabling interrupt) represents an interrupt enabling interval, and DI (disabling interrupt) represents an interrupt disabling interval.

It is assumed that, during execution of the routine **41** of the task A while the CPU **51** is in an interrupt enabling

condition, a system call to the OS **45** is issued at time **52**. As a result, the processing by the CPU **51** passes into the OS **45**, by which system call processing is thereafter performed. In order to perform such exclusive processing as described above in the processing in the OS **45**, interrupt disabling processing is executed at time **53** by the interrupt disabling processing means **46**. Consequently, an interruption disabling interval is started. The interrupt disabling processing is performed, for example, by executing an interrupt disabling instruction (e.g., "DI" instruction of some kinds of CPUs) of the CPU **51**.

Here, if the interrupt A (indicated as "INT A") from the interrupt factor **48** is inputted to the interrupt controller **50** at time **54** within the interrupt disabling interval in the system call processing, the interrupt controller **50** outputs an interrupt request to the CPU **51**. However, since the CPU **51** is within the interrupt disabling interval, the CPU **51** does not transfer the processing to the interrupt handler **44** but continues to execute the system call processing. Further, if the interrupt B (indicated as "INT B") is inputted from the interrupt factor **49**, which is another interrupt factor, at time **55** within the interrupt disabling interval, then the processing of the interrupt handler is held similarly as in the case of the interrupt A. The interrupt handler **43** does not generate a system call in the inside thereof at all, and even if processing by the interrupt handler **43** is executed during the system call processing, no conflict occurs with the resources managed by the OS **45**. Here, however, the interrupt processing by the interrupt handler **43** is held by the OS **45**.

After the processing for which exclusive control is required in the system call processing comes to an end, interrupt enabling processing is executed by the interrupt enabling processing means **47**. Here, it is assumed that the interrupt inhibiting processing comes to an end and an interrupt enabling interval is started at time **56**. Consequently, the interrupt handler **44** is executed only after the interrupt enabling processing is started, and the interrupt handler **43** is executed after completion of processing of the interrupt handler **44**. The interrupt enabling processing is performed, for example, by executing an interrupt enabling instruction (e.g., "EI" instruction) of the CPU **51**.

After all, the processing when an asynchronous interrupt event occurs in the present system is performed in such a manner as illustrated in FIG. **3**. In particular, the processing branches at step S15 depending upon whether the condition of the CPU **51** is the interrupt enabling condition (EI) or the interrupt disabling condition (DI). If the condition of the CPU **51** is the interrupt enabling condition, then the processing is transferred to a corresponding interrupt handler (step S16), and then, after the processing of the interrupt handler comes to an end, the processing returns to that executed prior to the interrupt (step S17), thereby completing the series of processes. On the other hand, if the condition of the CPU **51** is the interrupt disabling condition at step S15, interrupt handler processing is held until the interrupt enabling condition is entered subsequently (step S18). Then, after the interrupt enabling condition is entered, the processing advances to step S16 described above.

In the interrupt managing method by the conventional real-time OS described above, the OS collectively performs enabling/disabling processing for all interrupt factors. In particular, within an interrupt disabling interval, all interrupts are rejected. Consequently, a limitation by an interrupt disabling time arising from the OS exists equally to all interrupt factors, which is one of the obstacles to an improvement in operation efficiency of the system. Further, this limitation makes an obstacle to implement a real-time

OS into a system in which an interrupt for which no delay of the interrupt processing is allowed is present or another system in which a timer interrupt in a period shorter than a maximum value of the interrupt disabling time prescribed by the real-time OS is present. Thus the limitation is a cause of narrowing the range in application of the real-time OS.

Further, Japanese Patent Laid-Open Application No. Hei-2-220138 (JP. A. 2-220138), now abandoned, discloses a different technique wherein, when a system call is issued within a system call disabling interval in processing of an interrupt handler by an external interrupt, that is, within a software interrupt disabling interval, the system call processing is delayed until the system call disabling interval comes to an end, giving priority to the interrupt processing. With this technique, however, since an interrupt takes precedence over a system call only within a system call disabling interval explicitly designated in an interrupt handler, a limitation by an interrupt disabling time still exits.

SUMMARY OF THE INVENTION

It is an object of the present invention to provide an interrupt managing system wherein an interrupt which does not have an influence on the resource management by an OS is enabled even during a system call processing and an interrupt which does not issue a system call can be processed without any delay.

According to the present invention, the above object is achieved by an interrupt managing system for a computer system which includes a CPU and wherein resources are managed by a real-time operating system and an asynchronous interrupt to the CPU occurs, the interrupt managing system comprises managed interrupt storage means in which information regarding interrupts to be managed by the real-time operating system is stored, interrupt disabling processing means for reading out the information stored in the managed interrupt storage means and disabling an interrupt designated by the information in order to perform exclusive control regarding processing in the real-time operating system, and interrupt enabling processing means for enabling the disabled interrupt.

In the present invention, from among interrupts which are generated asynchronously, information of only those which issue a system call to the OS in a corresponding interrupt handler is stored into the managed interrupt storage means to manage those interrupts by the OS. Accordingly, any interrupt which does not directly relate to operation of the OS such as an interrupt in whose corresponding interrupt handler no system call is issued is not influenced by the OS at all, and consequently, processing relating to the interrupt can be performed without any delay. On the other hand, for any interrupt in whose corresponding interrupt handler a system call to the OS is issued, the same operation as in conventional systems is secured.

The above and other objects, features and advantages of the present invention will be apparent from the following description referring to the accompanying drawings which illustrate examples of preferred embodiments of the present invention.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is a block diagram showing the construction of a conventional computer system controlled by a typical real-time OS;

FIG. 2 is a timing chart illustrating processing in the computer system shown in FIG. 1 when interrupts successively occur;

FIG. 3 is a flow chart illustrating processing by the computer system shown in FIG. 1 when an interrupt occurs;

FIG. 4 is a block diagram showing the construction of a computer system to which an interrupt managing method of a first embodiment of the present invention is applied;

FIG. 5 is a timing chart illustrating processing by the computer system of the first embodiment when interrupts successively occur;

FIG. 6 is a flow chart illustrating processing by the computer system of the first embodiment when an interrupt occurs;

FIG. 7 is a flow chart illustrating processing by the computer system of the first embodiment when another interrupt occurs;

FIG. 8 is a block diagram showing the construction of a computer system to which an interrupt managing method of a second embodiment of the present invention is applied;

FIG. 9 is a timing chart illustrating processing by the computer system of the second embodiment when interrupts successively occur;

FIG. 10 is a flow chart illustrating processing by the computer system of the second embodiment when an interrupt occurs; and

FIG. 11 is a flow chart illustrating processing by the computer system OS of the second embodiment when another interrupt occurs.

DESCRIPTION OF THE PREFERRED EMBODIMENTS

First Embodiment

FIG. 4 shows a computer system to which a first embodiment of the present invention is applied. To a CPU 13, an interrupt controller 12 for transmitting occurrence of an interrupt to the CPU 13 and a main storage 1 are connected. In the present embodiment, it is assumed that the number of interrupt inputs which can be accepted by the interrupt controller 12 is sixteen (16) and the interrupt numbers from 0 to 15 are allocated to the respective interrupt inputs. Further, the interrupt controller 12 includes an interrupt mask table 30 for masking interrupts individually for the interrupt inputs, and when an input is received at one of those interrupt inputs which are not masked by the interrupt mask table 30, the interrupt controller 12 transmits occurrence of an interrupt to the CPU 13. A real-time OS 6 is resident in the main storage 1, and routines of tasks, routines of interrupt handlers are loaded into the main storage 1. Further, managed interrupt storage means 9 for storing and interrupt numbers managed by the OS 6 is provided within the main storage 1. Here, an interrupt A from an interrupt factor 10 and another interrupt B from another interrupt factor 11 are inputted to the interrupt controller 12. A routine 3 of a task A and another routine 4 of another task B are loaded as routines of tasks in the main storage 1, and an interrupt handler 4 started by the interrupt A and another interrupt handler 5 started by the interrupt B are loaded as interrupt handlers in the main storage 1. The managed interrupt storage means 9 is realized as a storage area which is referred to by the OS 6.

The OS 6 includes, in the inside thereof, interrupt disabling processing means 7 for inhibiting an interrupt of a particular interrupt number in order to perform exclusive control in system call processing, interrupt enabling processing means 8 for putting the CPU 13 into an interrupt enabling condition, and particular interrupt operation pro-

cessing means 14 for preventing acceptance by the CPU 13 of a particular one of interrupts disabled by the interrupt disabling processing means 7. More particularly, the interrupt disabling processing means 7 is a routine which is called to disable an interrupt and reads out interrupt numbers from the managed interrupt storage means 9. The interrupt enabling processing means 8 is a routine which manipulates the interrupt mask table 30 of the interrupt controller 12 via the CPU 13 and enables all interrupts. Meanwhile, the particular interrupt operation processing means 14 is a routine which performs writing into the interrupt mask table 30 of the interrupt controller 12 via the CPU 13 so that only interrupts of interrupt numbers read out from the managed interrupt storage means 9 by the interrupt disabling processing means 7 may be inhibited.

In the computer system of the present embodiment, interrupts inputted to the system are classified into two types. These are type I: interrupt which issues a system call to the OS 6 in an interrupt handler corresponding to the interrupt, and type II: interrupt which does not issue a system call in an interrupt handler corresponding to the interrupt. An interrupt of the type I is an interrupt for which management of enabling/disabling of the interrupt by the OS 6 is required, and the corresponding interrupt number is stored into the managed interrupt storage means 9. On the other hand, an interrupt of the type II does not issue a system call, and accordingly, enabling/disabling of the interrupt is not performed by the OS 6. The corresponding interrupt number is not stored in the managed interrupt storage means 9.

Here, it is assumed that the interrupts of the interrupt numbers 0 to 7 are of the type I and the interrupts of the interrupt numbers 8 to 15 are of the type II. The interrupt numbers 0 to 7 are stored in the managed interrupt storage means 9. Further, it is assumed that the interrupt A from the interrupt factor 10 is an interrupt of the type I whereas the interrupt B from the interrupt factor 11 is an interrupt of the type II. Accordingly, the interrupt handler 4 issues a system call in the inside of the routine thereof whereas the interrupt handler 5 does not issue a system call.

Now, the interrupt processing by the computer system is described with reference to FIG. 5. Here, description is given of the case wherein a system call to the OS 6 is issued by the task A during execution of the task A and, within the interrupt disabling interval in the system call processing, the interrupt A is inputted to the interrupt controller 12 from the interrupt factor 10 and then the interrupt B is inputted to the interrupt controller 12 from the interrupt factor 11. In FIG. 5, each thick line represents a routine being executed by the CPU 13. Further, EI (enabling interrupt) represents an interrupt enabling interval, and DI (disabling interrupt) represents an interrupt disabling interval.

It is assumed that, during execution of the routine 2 of the task A while the CPU 13 is in the interrupt enabling condition for all interrupts, a system call to the OS 6 is issued at time 15. As a result, the processing by the CPU 13 is transferred from the task A into the OS 6, by which system call processing is performed. During the system call processing, since disabling of any interrupt must be performed within an interval within which exclusive control for the resources is required, the interrupt disabling processing means 7 is called at time 16. The interrupt disabling processing means 7 reads out the interrupt numbers stored in the managed interrupt storage means 9 and outputs the interrupt numbers thus read out as interrupts to be disabled to the particular interrupt operation processing means 14. The particular interrupt operation processing means 14 operates the interrupt mask table 30 of the interrupt controller 12 so that only an interrupt of the type I may not be accepted. Consequently, any interrupt of the type I, that is, the interrupt A by the interrupt factor 10, is inhibited. In this instance, an interrupt of the type II is not disabled.

Here, it is assumed that the interrupt A (indicated as "INT A") from the interrupt factor 10 occurs at time 17 within the interrupt disabling period in the system call processing. Here, since interrupt disabling is set for the interrupt A, the interrupt controller 12 holds the interrupt A. Accordingly, the processing of the CPU 13 does not pass to the interrupt handler 4.

It is assumed that the interrupt B (indicated as "INT B") from the interrupt factor 11 is subsequently generated at time 18 within the interrupt disabling interval. As described above, the interrupt B does not issue a system call in the corresponding interrupt handler 5 and is excepted from the object of management by the OS 6 and accordingly is enabled. Therefore, the interrupt B suspends the system call processing without being influenced by the OS 6 and the interrupt handler 5 is started immediately.

After completion of the processing of the interrupt handler 5, the processing passes back to the suspended system call processing, and an interrupt of the type I is enabled by the interrupt enabling processing means 8 at time 19. As a result, the interrupt A held formerly is accepted, and the processing is transferred to the interrupt handler 4.

FIG. 6 is a flow chart illustrating processing when an interrupt of the type I occurs. If an interrupt of the type I is inputted, then the processing branches at step S1 depending upon whether the CPU 13 is in the interrupt enabling condition (EI) or the interrupt disabling condition (DI). If the CPU 13 is in the interrupt enabling condition, that is, the interrupt is not masked at the interrupt mask table 30, then the processing passes to a corresponding interrupt handler (step S2), and then after the processing of the interrupt handler comes to an end, the processing returns to the processing which has been executed prior to the interrupt (step S3), thereby completing a series of processes. On the other hand, if the CPU 13 is in the interrupt disabling condition, that is, the interrupt is masked by the interrupt mask table 30, then the interrupt handler processing is held until the interrupt enabling condition is entered again (step S4). After the interrupt enabling condition is entered again, the processing advances to step S2 described above. After all, similar processing to that performed using the conventional interrupt managing method is performed.

On the other hand, FIG. 7 is a flow chart illustrating processing when an interrupt of the type II occurs. If an interrupt of the type II is inputted, then branching of the processing depending upon whether the CPU 13 is in the interrupt enabling condition (EI) or the interrupt disabling condition (DI) does not occur as seen from step S5. Then, in any case, the processing is transferred to a corresponding interrupt handler at step S6. Then, after the processing of the interrupt handler comes to an end, the processing returns to the processing which has been performed prior to the interrupt (step S7), thereby completing a series of processes. After all, the interrupt processing is executed without being influenced by the OS.

In this manner, in the computer system which employs the interrupt managing method of the present embodiment, interrupt processing for which high emergency is required can be executed immediately without being influenced by the OS if it does not issue a system call to the OS. Accordingly, a real-time OS can be implemented into fields into which implementation is obstructed in conventional

processing because the interrupt disabling time by an OS is long. Further, when it is required to issue a system call to the OS from within an interrupt handler, quite similar processing to conventional processing is performed, and consequently, the compatibility with conventional systems is maintained.

Second Embodiment

FIG. **8** shows a computer system to which a second embodiment of the present invention is applied. The computer system of the second embodiment is similar to the computer system of the first embodiment, but is different in that it includes a timer controller **20** which generates a timer interrupt C in place of the interrupt factor which outputs the interrupt B, and a timer interrupt handler **21** is prepared in place of the interrupt handler **5** in the main storage **1** and managed interrupt information reception means **22** is provided in the OS **6**. The timer interrupt handler **21** is an interrupt handler whose interrupt factor is the timer interrupt C from the timer controller **20**, and does not issue a system call to the OS **6** and executes interrupt processing for which no delay is allowed. Meanwhile, the managed interrupt information reception means **22** receives information of an interrupt to be managed by the OS **6** from an application program and stores a corresponding interrupt number into the managed interrupt storage means **9**.

Next, the processing by the computer system is described with reference to FIG. **5**. Here, it is assumed that the interrupt handler A corresponding to the interrupt A issues a system call.

First, as setting of the OS **6** upon initialization, only the interrupt A is set to be managed by the OS **6**. This is because, in the application program, a system call to the OS **6** is issued only from within an interrupt handler **2** which is started in response to the interrupt A. This setting can be modified by the application program. The OS **6** receives, upon initialization, information to be stored into the managed interrupt storage means **9** from the application program by the managed interrupt information reception means **22**. Here, the interrupt number corresponding to the interrupt A is received and stored into the managed interrupt storage means **9**.

In the following, description is given of the case wherein, after registration into the managed interrupt storage means **9** is performed in this manner, a system call to the OS **6** is issued by the task A during execution of the task A, and the interrupt A is inputted from the interrupt factor **10** to the interrupt controller **12** and then the timer interrupt C occurs within the interrupt disabling interval in the system call processing.

It is assumed that the CPU **13** is in the interrupt enabling condition for all interrupts, and during execution of the routine **2** of the task A, a system call to the OS **6** is issued at time **35**. As a result, the processing by the CPU **13** is transferred into the OS **6**, by which system call processing is performed. During the system call processing, since interrupt must be disabled within an interval within which exclusive control of the resources is required, the interrupt disabling processing means **7** is called at time **36**. The interrupt disabling processing means **7** reads out the interrupt numbers stored in the managed interrupt storage means **9**. Since only the interrupt number corresponding to the interrupt A is stored in the managed interrupt storage means **9** as a result of setting of the OS **6** upon initialization, only this interrupt number is transferred to the particular interrupt operation processing means **14**. The particular interrupt operation processing means **14** operates the interrupt mask table **30** of the interrupt controller **12** based on the thus transferred interrupt number so that only the interrupt A is disabled.

Here, it is assumed that the interrupt A (indicated as "INT A") from the interrupt factor **10** is inputted at time **37** within the interrupt disabling interval in the system call processing. Since the interrupt disabling for the interrupt A is set, the interrupt controller **12** holds the interrupt A. Accordingly, the processing of the CPU **13** does not is not transferred to the interrupt handler **4**.

Then, it is assumed that the timer interrupt C (indicated as "CLOCK INT") is generated by the timer controller **20** at time **38** within the interrupt disabling interval. Since the timer interrupt C is not placed under the management of the OS **6** as a result of setting of the OS **6** upon initialization, the timer interrupt C is enabled. Consequently, even when the CPU **13** is within the interrupt disabling interval, the system call processing is suspended by the timer interrupt C and the timer interrupt handler **21** is started immediately.

After completion of the processing of the timer interrupt handler **21**, the processing returns to the interrupted system call processing, and at time **39**, the interrupt A is enabled by the interrupt enabling processing means **8**. As a result, the interrupt A held precedently is accepted, and the processing passes to the interrupt handler **4**.

FIG. **10** is a flow chart illustrating the processing when the interrupt A occurs. When the interrupt A is inputted, the processing branches at step S8 depending upon whether the condition of the CPU **13** is the interrupt enabling condition (EI) or the interrupt disabling condition (DI). If the condition of the CPU **13** is the interrupt enabling condition, that is, the interrupt A is not masked at the interrupt mask table **30**, then the processing is transferred to the interrupt handler **4** (step S9), and after the processing of the interrupt handler **4** comes to an end, the processing returns to the processing which has been performed prior to the interrupt (step S10), thereby completing a series of processes. On the other hand, if the condition of the CPU **13** is the interrupt disabling condition, that is, the interrupt A is masked by the interrupt mask table **30**, the processing of the interrupt handler **4** is held until after the interrupt enabling condition is entered (step S11). After the interrupt enabling condition is entered, the processing is transferred to step S9 described above. After all, processing similar to that performed using the conventional interrupt managing method is performed.

Meanwhile, FIG. **11** is a flow chart illustrating the processing when the timer interrupt C is generated. When the timer interrupt C is inputted, branching of the processing depending upon whether the condition of the CPU **13** is the interrupt enabling condition (EI) or the interrupt disabling condition (DI) does not occur as seen from step S12. In any case, the processing is transferred to the timer interrupt handler **21** (step S13), and after the processing of the timer interrupt handler **21** comes to an end, the processing returns to that performed prior to the interrupt (step S14), thereby completing a series of processes. After all, the timer interrupt C is processed without being influenced by the OS.

In the present embodiment, a timer interrupt which has a high degree of urgency is processed without being influenced by the OS. For example, in a system which establishes synchronism of communication based on clocks produced by the timer controller **20**, clock interrupts at accurate intervals are essentially required. Also to such a field that attaches importance to the accuracy in interrupt starting interval time, introduction of a real-time OS can be achieved

by making use of the present embodiment. Further, where there is the necessity to issue a system call to an OS from within an interrupt handler, operation quite similar to that performed by the conventional interrupt managing method can be performed, and accordingly, the compatibility with conventional systems is maintained.

It is to be understood that variations and modifications of the interrupt managing system disclosed herein will be evident to those skilled in the art. It is intended that all such modifications and variations be included within the scope of the appended claims.

What is claimed is:

1. In a computer system which includes a CPU, wherein resources are managed by a real-time operating system and an asynchronous interrupt to the CPU may occur, an interrupt managing system for said computer, comprising:

managed interrupt storage means in which information regarding interrupts to be managed by the real-time operating system is stored; interrupt disabling processing means for reading out the information stored in the managed interrupt storage means and disabling an interrupt designated by the information in order to perform exclusive control regarding processing in the real-time operating system; and

interrupt enabling processing means for enabling the disabled interrupt,

wherein interrupts that are not to be managed by the real-time operating system are not disabled by the interrupt disabling processing means, and are executed without delay by the CPU.

2. The interrupt managing system according to claim 1, further comprising:

an interrupt controller connected to the CPU and having a plurality of individually maskable interrupt inputs for transmitting an interrupt to the CPU in response to an input to one of the maskable interrupt inputs which is not masked; and

particular interrupt operation means for setting a mask to the interrupt controller corresponding to the interrupt disabled by the interrupt disabling processing means.

3. In a computer system which includes a CPU, wherein resources are managed by a real-time operating system and an asynchronous interrupt to the CPU may occur, an interrupt managing system for said computer, comprising:

managed interrupt storage means in which information regarding interrupts to be managed by the real-time operating system is stored;

interrupt disabling processing means for reading out the information stored in the managed interrupt storage means and disabling an interrupt designated by the information in order to perform exclusive control regarding processing in the real-time operating system;

interrupt enabling processing means for enabling the disabled interrupt;

an interrupt controller connected to the CPU and having a plurality of individually maskable interrupts for transmitting an interrupt to the CPU in response to an input to one of the maskable interrupt inputs which is not masked; and

particular interrupt operation means for setting a mask to the interrupt controller corresponding to the interrupt disabled by the interrupt disabling processing means,

wherein only information regarding an interrupt in whose corresponding interrupt handler a system call to the real-time operating system is issued is stored in the managed interrupt storage means.

4. The interrupt managing system according to claim 2, wherein the interrupt enabling processing means cancels a mask set to the interrupt controller.

5. The interrupt managing system according to claim 2, wherein the interrupt disabling processing means, the interrupt enabling processing means and the particular interrupt operation means are constructed as individual routines in the real-time operating system.

6. In a computer system which includes a CPU, wherein resources are managed by a real-time operating system and an asynchronous interrupt to the CPU may occur, an interrupt managing system for said computer, comprising:

managed interrupt storage means in which information regarding interrupts to be managed by the real-time operating system is stored;

interrupt disabling processing means for reading out the information stored in the managed interrupt storage means and disabling an interrupt designated by the information in order to perform exclusive control regarding processing in the real-time operating system;

interrupt enabling processing means for enabling the disabled interrupt;

an interrupt controller connected to the CPU and having a plurality of individually maskable interrupts for transmitting an interrupt to the CPU in response to an input to one of the maskable interrupt inputs which is not masked; and

particular interrupt operation means for setting a mask to the interrupt controller corresponding to the interrupt disabled by the interrupt disabling processing means,

wherein a timer controller which generates a timer interrupt is connected to the interrupt controller, and information regarding the timer interrupt is not stored in the managed interrupt storage means.

7. The interrupt managing system according to claim 6, wherein a system call to the real-time operating system is not issued in a timer interrupt handler corresponding to the timer interrupt.

8. In a computer system which includes a CPU, wherein resources are managed by a real-time operating system and an asynchronous interrupt to the CPU may occur, an interrupt managing system for said computer, comprising:

managed interrupt storage means in which information regarding interrupts to be managed by the real-time operating system is stored;

interrupt disabling processing means for reading out the information stored in the managed interrupt storage means and disabling an interrupt designated by the information in order to perform exclusive control regarding processing in the real-time operating system;

interrupt enabling processing means for enabling the disabled interrupt;

an interrupt controller connected to the CPU and having a plurality of individually maskable interrupts for transmitting an interrupt to the CPU in response to an input to one of the maskable interrupt inputs which is not masked;

particular interrupt operation means for setting a mask to the interrupt controller corresponding to the interrupt disabled by the interrupt disabling processing means,

managed interrupt information reception means for storing managed interrupt information received from an application program into the managed interrupt storage means, the managed interrupt information reception means being constructed as an individual routine in the real-time operating systems

wherein the interrupt disabling processing means, the interrupt enabling processing means and the particular

interrupt operation means are connected as individual routines in the real-time operating system.

9. The interrupt managing system according to claim 8, wherein, upon initialization of the real-time operating system, the managed interrupt information reception means receives and stores the managed interrupt information into the managed interrupt storage means.

10. The interrupt managing system according to claim 1, wherein the interrupts that are not disabled by the disabling processing means correspond to interrupts that do not issue a system call in a respective interrupt handler corresponding to the interrupts.

* * * * *

US005995745A

United States Patent [19]

Yodaiken

[11] Patent Number: 5,995,745

[45] Date of Patent: Nov. 30, 1999

[54] **ADDING REAL-TIME SUPPORT TO GENERAL PURPOSE OPERATING SYSTEMS**

[76] Inventor: **Victor J. Yodaiken**, P.O. Box 638, Socorro, N.Mex. 87801

[21] Appl. No.: **08/967,146**

[22] Filed: **Nov. 10, 1997**

Related U.S. Application Data

[60] Provisional application No. 60/033,743, Dec. 23, 1996.

[51] **Int. Cl.**[6] **G06F 9/455**

[52] **U.S. Cl.** **395/500.47**; 395/500.43; 709/103; 710/262

[58] **Field of Search** 395/500, 670, 395/677, 733, 735, 500.43, 500.44, 500.47; 709/100, 102, 103, 107; 710/260, 262

[56] **References Cited**

U.S. PATENT DOCUMENTS

4,553,202	11/1985	Trufyn	710/269
4,993,017	2/1991	Bachinger et al.	370/360
5,291,614	3/1994	Baker et al.	712/35
5,721,922	2/1998	Dingwall	709/103

OTHER PUBLICATIONS

H. Lycklama et al., "UNIX Time-Sharing System: The MERT Operating System," The Bell System Technical Journal, vol. 57, No. 6, pp. 2049–2086, Jul.–Aug. 1978.

Primary Examiner—Kevin J. Teska
Assistant Examiner—Russell W. Frejd
Attorney, Agent, or Firm—Ray G. Wilson

[57] **ABSTRACT**

A general purpose computer operating system is run using a real time operating system. A real time operating system is provided for running real time tasks. A general purpose operating system is provided as one of the real time tasks. The general purpose operating system is preempted as needed for the real time tasks and is prevented from blocking preemption of the non-real time tasks.

11 Claims, 6 Drawing Sheets

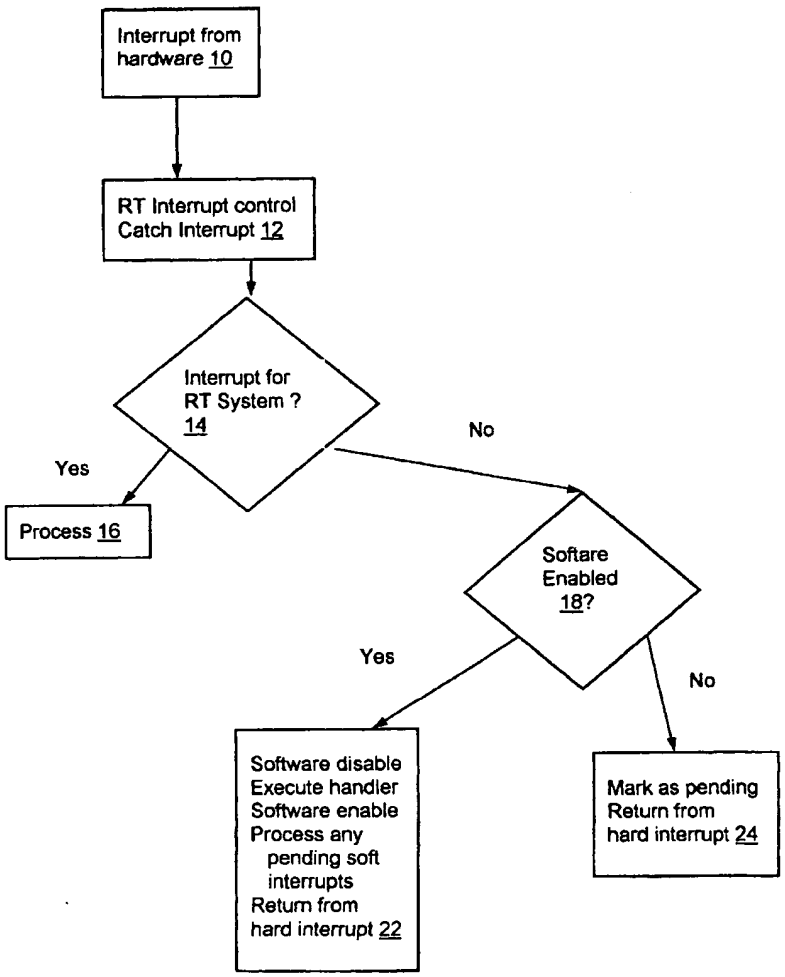


FIGURE 1A

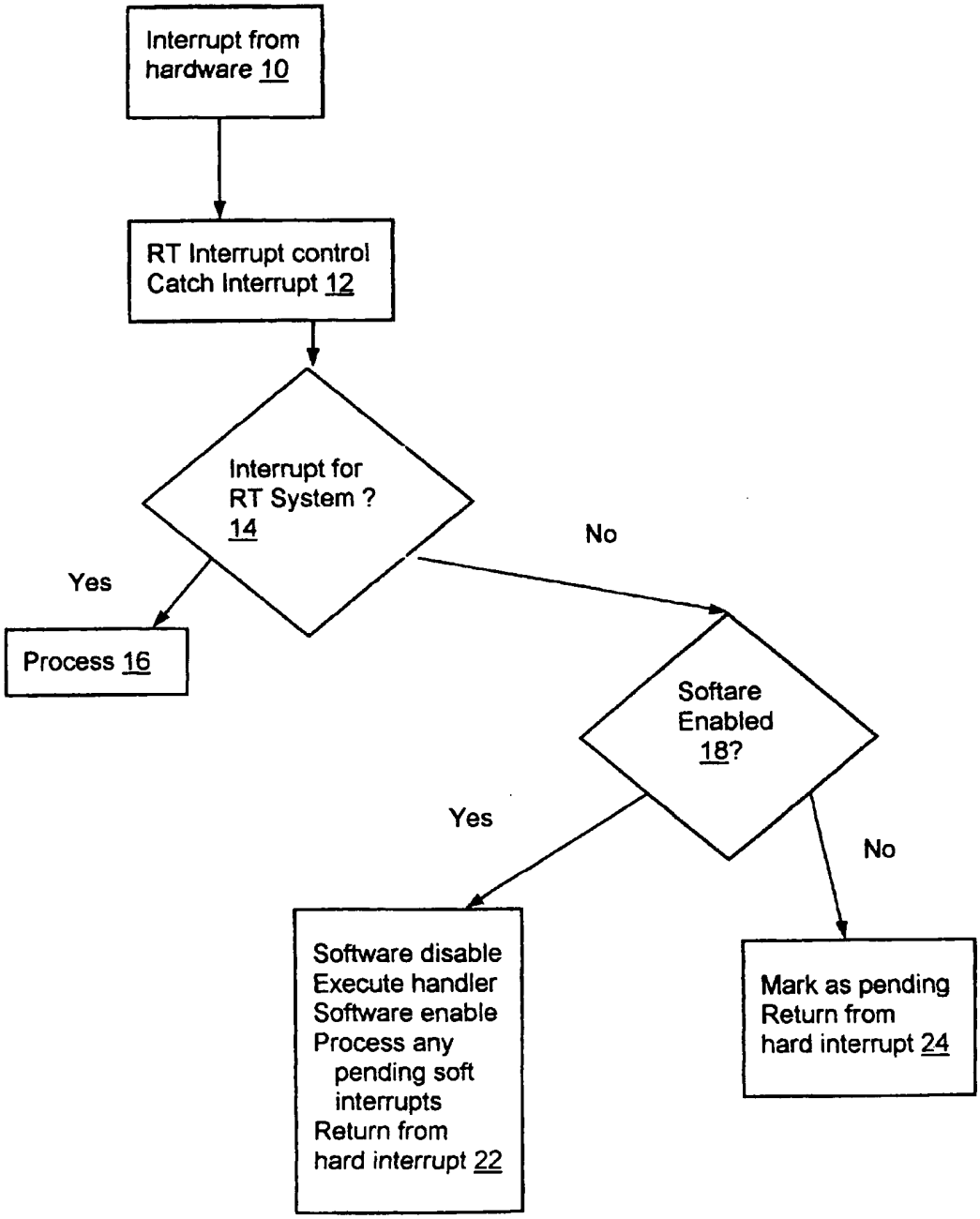
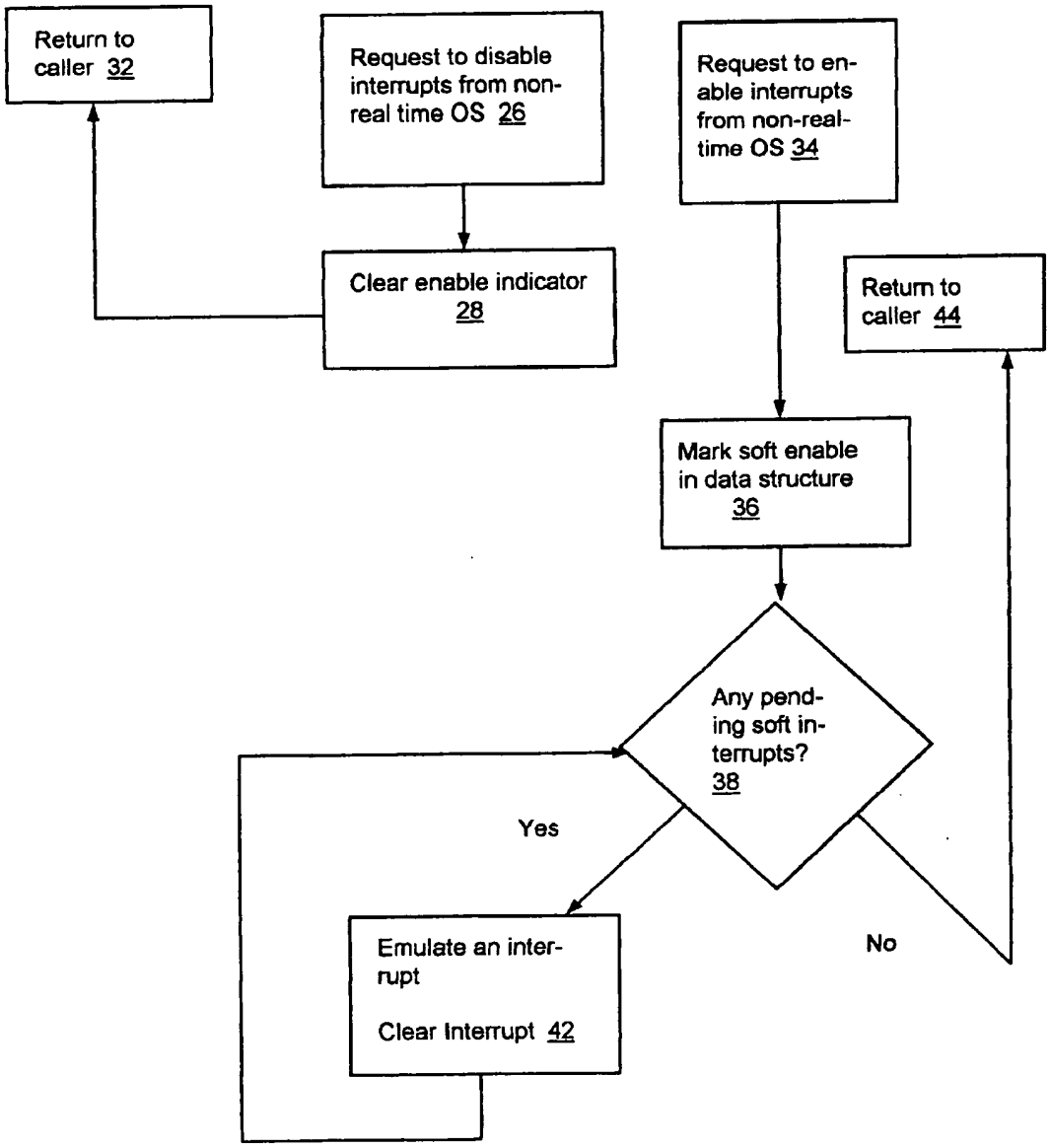

Interrupt from hardware 10
RT Interrupt control Catch Interrupt 12
Interrupt for RT System ? 14
Yes
No
Process 16
Softare Enabled 18?
Yes
No
Software disable
Execute handler
Software enable
Process any pending soft interrupts
Return from hard interrupt 22
Mark as pending
Return from hard interrupt 24

FIGURE 1B


Return to caller 32
Request to disable interrupts from non-real time OS 26
Request to enable interrupts from non-real-time OS 34
Clear enable indicator 28
Return to caller 44
Mark soft enable in data structure 36
Any pending soft interrupts? 38
Yes
No
Emulate an interrupt
Clear Interrupt 42

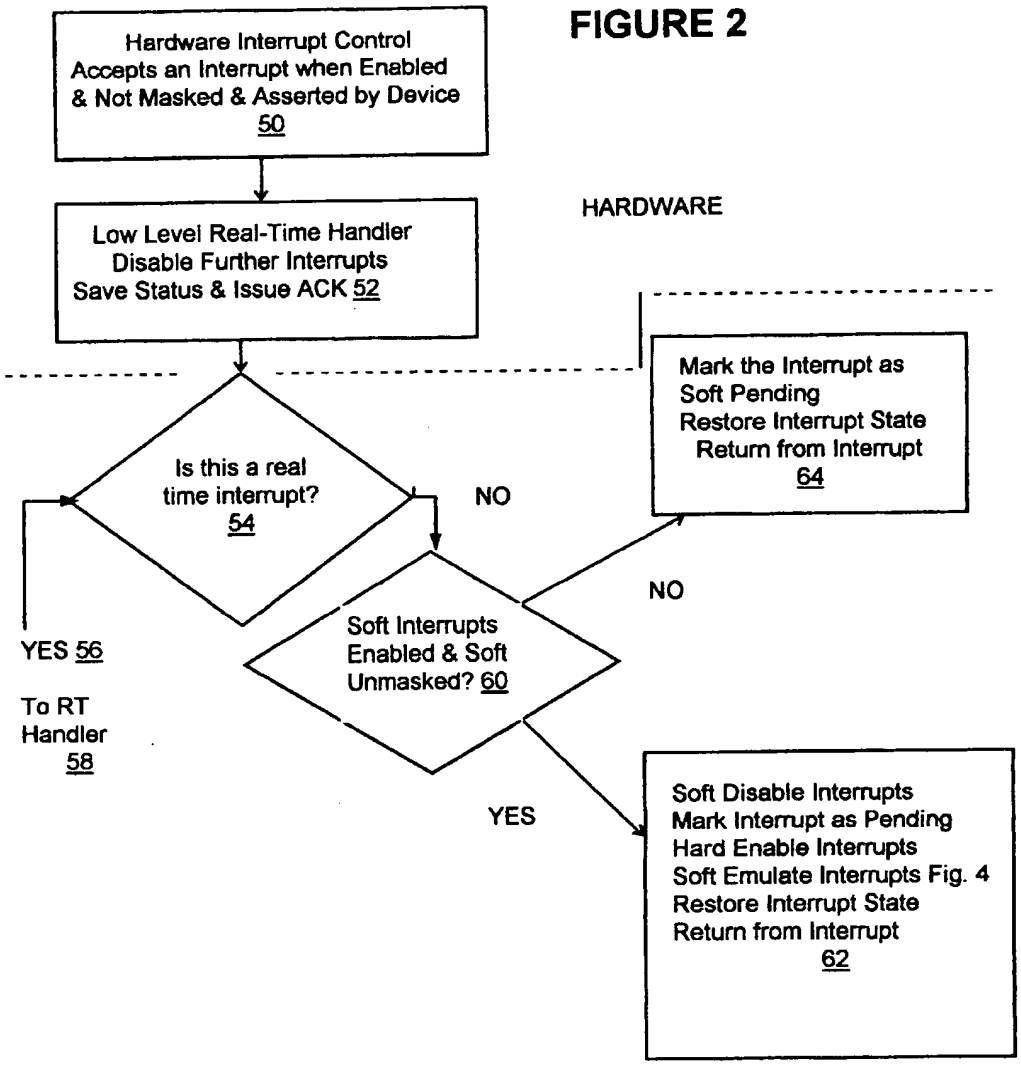

Hardware Interrupt Control
Accepts an Interrupt when Enabled
& Not Masked & Asserted by Device
50
HARDWARE
Low Level Real-Time Handler
Disable Further Interrupts
Save Status & Issue ACK 52
Mark the Interrupt as
Soft Pending
Restore Interrupt State
Return from Interrupt
64
Is this a real
time interrupt?
54
NO
NO
YES 56
To RT
Handler
58
Soft Interrupts
Enabled & Soft
Unmasked? 60
YES
Soft Disable Interrupts
Mark Interrupt as Pending
Hard Enable Interrupts
Soft Emulate Interrupts Fig. 4
Restore Interrupt State
Return from Interrupt
62

FIGURE 2

FIGURE 3

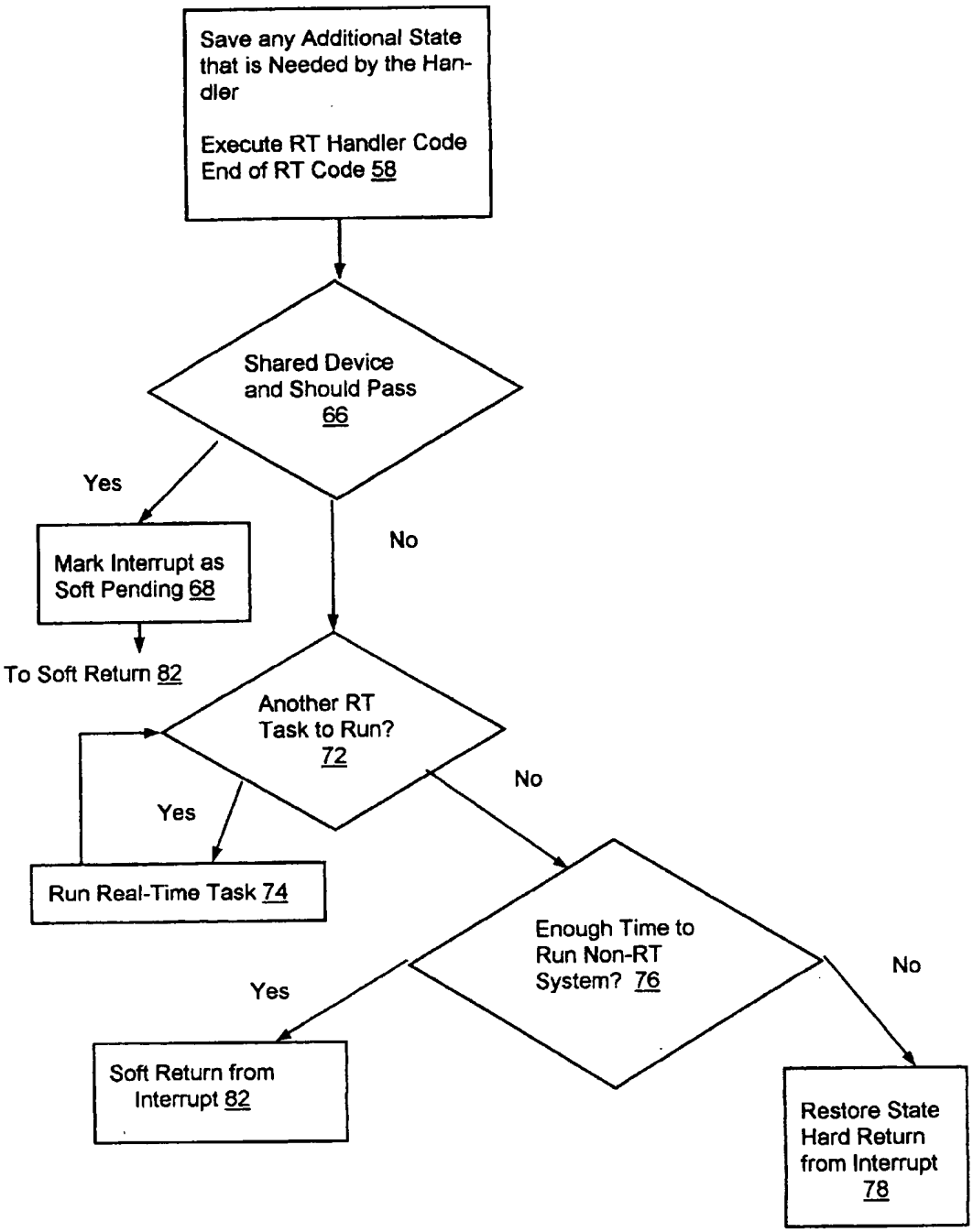

Save any Additional State that is Needed by the Handler
Execute RT Handler Code End of RT Code 58
Shared Device and Should Pass 66
Yes
No
Mark Interrupt as Soft Pending 68
To Soft Return 82
Another RT Task to Run? 72
Yes
No
Run Real-Time Task 74
Enough Time to Run Non-RT System? 76
Yes
No
Soft Return from Interrupt 82
Restore State Hard Return from Interrupt 78

FIGURE 4

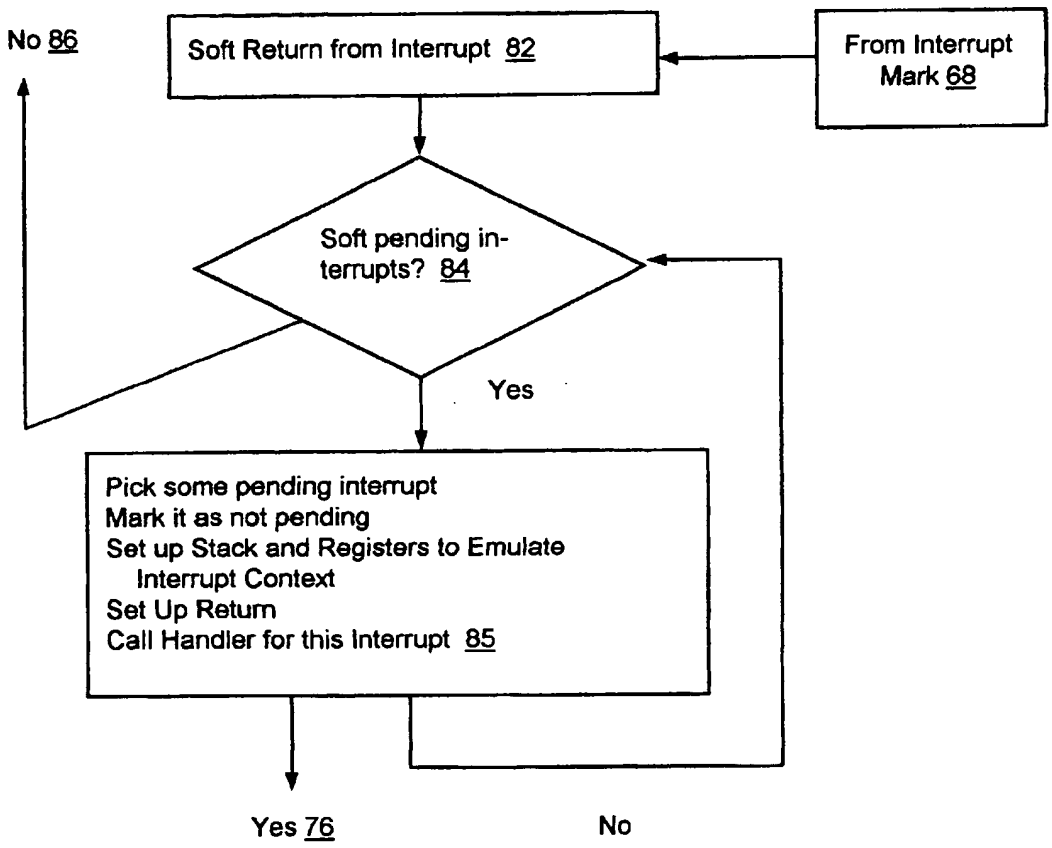

No 86
Soft Return from Interrupt 82
From Interrupt Mark 68
Soft pending interrupts? 84
Yes
Pick some pending interrupt
Mark it as not pending
Set up Stack and Registers to Emulate Interrupt Context
Set Up Return
Call Handler for this Interrupt 85
Yes 76
No

FIGURE 5

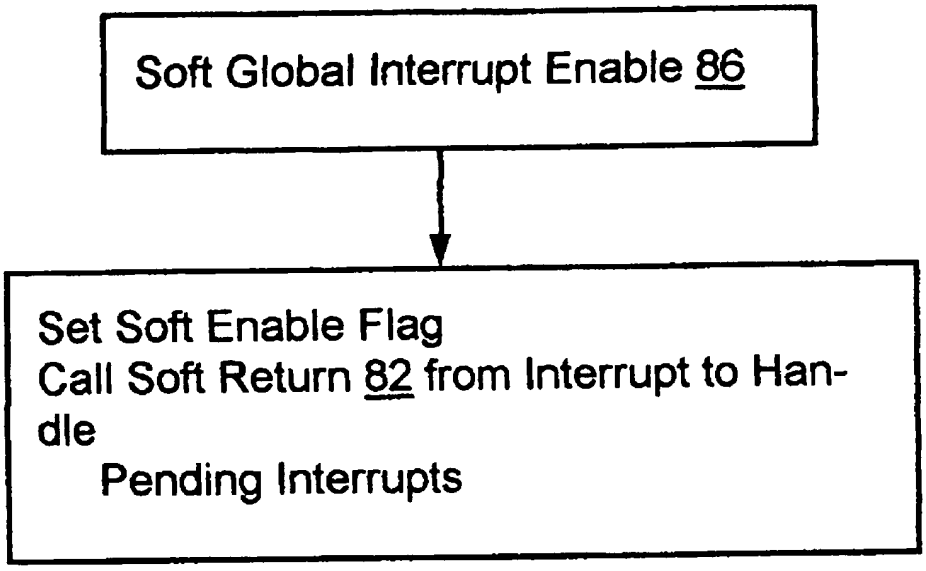
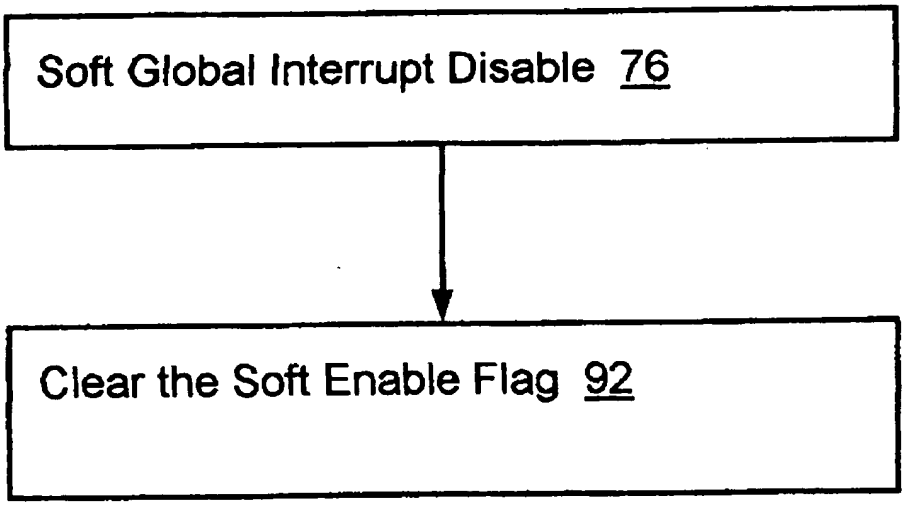


FIGURE 6



ADDING REAL-TIME SUPPORT TO GENERAL PURPOSE OPERATING SYSTEMS

This application claims the benefit of U.S. provisional application Ser. No. 60/033,743 filed Dec. 23, 1996.

BACKGROUND OF THE INVENTION

This invention relates to real-time operating systems, and, more particularly, to running a real-time operating system with a general purpose operating system.

General purpose computer operating systems "time-share" programs to optimize use of the machine or response time, or some other mix of objectives. Such programs cannot be used for control of instruments, robots, machine tools, communication devices, or other machinery that requires operation at "hard" real-time, i.e., precise timing where deadlines cannot be missed, because the systems are designed to optimize average performance of application programs at the expense of predictability. Performing the above tasks requires real-time support in the operating system: an ability to schedule tasks at precise intervals, no matter what other system tasks may be active. But real-time support can only be offered if the operating system can ensure (1) low interrupt latency (fast response) and (2) fully pre-emptive task switching. Low interrupt latency means that whenever a hardware device or the clock signals the processor, the operating system will execute a "handler" program within a short and bounded interval of time. Fully preemptive task switching means that whenever a high-priority real-time task is scheduled, it can be executed no matter what other tasks are currently executing.

An exemplary need is a controller for an instrument that measures electrical discharges in thunderstorms. It is desirable to read data from the instruments periodically, buffer and then write the data to disk, generate a graphical display of the data either locally or over the network, and possibly accept data from other instruments over the network. Only the first of these tasks requires hard real-time; the remainder are standard programming tasks for which a general purpose operating system is well suited.

Another exemplary need is the control of a liquid fueled rocket mounted on a test platform. There is a need to sample and display data on numerous channels, update a remote real-time display, accept emergency shutdown commands, and perform routing control operations. Again, most of the requirements are for conventional operating systems services, but there are hard real-time components that need reasonably precise scheduling. For example, the shutdown sequence must be precisely timed and cannot be delayed by lower priority tasks without spectacular and unwelcome results.

It is possible to design a special purpose operating system to support real-time, but this is an enormously complex, expensive and error-prone process that produces a system that needs a large continuing investment to remain current. In addition, there is substantial ongoing development effort and it is desirable to maintain compatibility with these developments, which are generally done by others.

Accordingly, it is an object of the present invention to operate a real-time operating system, or executive, and retain the capabilities offered by a general purpose operating system.

Additional objects, advantages and novel features of the invention will be set forth in part in the description which follows, and in part will become apparent to those skilled in the art upon examination of the following or may be learned by practice of the invention. The objects and advantages of the invention may be realized and attained by means of the instrumentalities and combinations particularly pointed out in the appended claims.

SUMMARY OF THE INVENTION

To achieve the foregoing and other objects, and in accordance with the purposes of the present invention, as embodied and broadly described herein, this invention comprises a process for running a general purpose computer operating system using a real time operating system. A real time operating system is provided for running real time tasks. A general purpose operating system is provided as one of the real time tasks. The general purpose operating system is preempted as needed for the real time tasks.

BRIEF DESCRIPTION OF THE DRAWINGS

The accompanying drawings, which are incorporated in and form a part of the specification, illustrate embodiments of the present invention and, together with the description, serve to explain the principles of the invention. In the drawings:

FIG. **1A** is a software flow diagram for the process of the present invention.

FIG. **1B** is a software flow diagram for additional routines used in the flow diagram shown in FIG. **1A**

FIG. **2** is a software flow diagram for application of the present invention to the LINUX operating system.

FIG. **3** is a continuation of the software flow diagram shown in FIG. **2**.

FIG. **4** is a continuation of the software flow diagram shown in FIG. **2**.

FIG. **5** is a continuation of the software flow diagram shown in FIG. **4**.

FIG. **6** is a continuation of the software flow diagram shown in FIG. **4**.

DETAILED DESCRIPTION

In accordance with the present invention, a real-time operating system, hereinafter called a RT-executive, runs a general purpose operating system as its lowest priority task, preempting it when needed. All operating systems have routines that enable and disable interrupts (or set interrupt levels). These routines are "captured" so that they pass through a small RT-executive and hardware emulator.

Referring first to FIG. **1A**, when the general purpose operating system attempts to disable hardware interrupts in accordance with the present invention, the emulator sets an indicator in software. When interrupts arrive **10**, they are captured **12** by the RT-executive. The nature of the interrupt is then determined **14**. If the interrupts cause a real-time task to be runnable, that task is started **16**. If the interrupts are passed through to the general purpose operating system, the program determines if the software enable indicator is set **18**. If the software enable indicator is set, then an interrupt is emulated **22** and the handler in the general purpose operating system is run. A handler is a software routine that processes a particular system interrupt. If the enabled indicator is not set **24** (i.e., is cleared), a software pending interrupt indicator is set. As conventionally used herein, the term "soft" interrupt is used to mean interrupts that are emulated in software by the RT-executive; the term "hard" interrupt means interrupts generated by hardware.

As seen in FIG. **1B**, when the general purpose operating system attempts to enable hardware interrupts **34**, the

enabled indicator is set **36** and the pending indicator is checked **38**. If an interrupt is pending, than an interrupt is emulated **42**. If no software interrupts are pending, the request is returned **44**. If the general purpose operating system attempts to disable hardware interrupts **26**, the enabled indicator is cleared **28** and the request is returned **32**. The result is that the real-time tasks are never disabled from the executing, while the general purpose operating system requires only very minimal modifications.

Thus, a form of emulator is the following routines, where the general purpose operating system uses the routines disable and enable to turn interrupts on and off, and also uses a routine mask to selectively enable and disable interrupts per/device:

```
Soft_Enable 1:  sets a flag indicating that soft interrupts are enabled
Soft_Enable 2:  sets a flag indicating that interrupts are enabled and
                then processes pending soft interrupts.
Soft_Disable:   sets a flag indicating that soft interrupts are disabled
Soft_MaskOff:   passes a device or interrupt number, sets a flag to
                disable soft interrupts from that device or interrupt
                number
Soft_MaskOn     passes a device or interrupt number, sets a flag to
                enable soft interrupts from that device or interrupt
                number
LowLevelHandler (per hardware interrupt):

    If interrupt is handled by real time handler
        Then pass control to that handler
    When the real-time handler, if any, completes operation
      If interrupt is handled by soft handler or is shared
          AND no realtime task if active
            AND soft interrupts are enabled
    Then mark soft interrupts disabled and pass control to soft handler
    Else save interrupt information and set PendingFlag
```

This routine works with minor modifications for systems that set processor levels instead of simply turning interrupts on and off.

In a particular embodiment of the present invention as discussed below, the process is applied to the Linux operating system, a UNIX-derivative operating system that is publicly available from a variety of sources including numerous internet sites. See, e.g., http://www.ssc.com/linux/resources/apps.html. As used herein, Linux interacts with a software emulation of the interrupt control hardware. The emulation supports the synchronization requirements of the Linux kernel while preventing Linux from disabling interrupts. Interrupts that are handled by Linux are passed through to the emulation software after any needed real-time processing completes. If Linux has requested that interrupts be disabled, the emulation software simply marks the interrupts as pending. When Linux requests that interrupts be enabled, the emulation software causes control to switch to the Linux handler for the highest priority pending interrupt. Linux is then able to provide sophisticated services to the real-time system without increasing interrupt latency.

A virtual machine layer has been advanced as a technique for making UNIX real-time as far back as 1978 (H. Lycklama et al., "Unix time-sharing system: The MERT operating system," Bell System Technical Journal, 57(6) :2049–2086 (1978)). But the present system emulates only a specific hardware component—interrupt control. Linux is able to otherwise directly control the hardware both for run-time efficiency and in order to minimize the need for modifications to the Linux kernel. The real-time executive that acts as the 0-level operating system does not provide any basic services that can be provided by Linux. Instead, the real-time executive is intended to provide services that Linux cannot provide. Thus, the real-time executive does not provide network services or virtual memory or access to a file system.

One goal of this invention was to develop a Linux kernel that would support real-time control of scientific instruments. The limitations of standard time-shared operating systems for this purpose include unpredictability of execution and high interrupt latency. General purpose time-shared operating systems have schedulers that are intended to balance response time and throughput. As a result, the execution of any process depends in a complex and unpredictable fashion on system load and the behavior of other processes. These problems are compounded in Linux, and most other UNIX derivatives, because kernel mode operation is non-preemptable and because disabling interrupts is used as the primary means of synchronization.

Low interrupt handling latency is critical for any real-time operating system. But interrupt latency is high in Linux. On a 120 MHz Pentium-based PC (Pentium is a trademark of Intel Corp.), up to 400 μsec latency has been measured for handling "fast" Linux interrupts. It has been reported that the Linux console driver disables interrupts for as long as several milliseconds when switching virtual consoles. Clearly, a frame-buffer that must be emptied every millisecond is then beyond the capabilities of the system, and this timing requirement is among the least demanding.

The fundamental limits for real-time processing are determined by the hardware. For example, a test system for running the present system requires a time of approximately 3.2 μsec for setting a bit on the parallel port. Obviously, this does not support a requirement for a data rate of over 280 KHz, regardless of the operating system. Similarly, the minimal interrupt latency is bounded by the hardware interrupt processing time. On a Pentium processor, at least 61 cycles are needed to enter and exit the interrupt, and some time is also needed for the interaction with the interrupt controller. Devices that need more rapid response or more precise timing call for dedicated, or at least different, hardware. But modem PC hardware is capable of handling the real-time requirements of a wide range of devices.

The current version of RT-Linux is a modification of Linux 2.1 and 2.0 for Intel x86 based uni-processors and multi-processors. Efforts are currently underway to move to a 2.0 Linux kernel and to port the system to other processor architectures, including the IBM/Motorola PowerPC and DEC Alpha. The test system has a 120 MHz Pentium processor, a 512 KB secondary cache and 32 MB of main memory. All I/O devices, other than the video display and keyboard are DMA devices. Non-DMA controllers for mass storage devices are difficult to integrate into a real-time control system.

A simple priority-based preemptive scheduler is currently used in RT-Linux. It is implemented as a routine that chooses among the ready process the highest-priority one and marks it as a next process to execute. Tasks give up the processor voluntarily, or are preempted by a higher priority task when its time to execute comes.

Typically, there is a tradeoff between the clock interrupt rate and the task release jitter. In most systems, tasks are resumed in the periodic clock interrupt handler. High clock interrupt rate ensures low jitter, but, at the same time, incurs much overhead. Low interrupt rate causes tasks to be resumed either too early or too late. In RT-Linux this tradeoff is resolved by using a one-shot timer instead of a periodic clock. Tasks are resumed in the timer interrupt handler precisely when needed.

Note that all task resources are statically defined. In particular, there is no support for dynamic memory allocation. The basic approach is that any sophisticated services that require dynamic memory allocation would be moved

into Linux processes. In keeping with this approach the real-time kernel itself is not preemptable.

Since the Linux kernel can be preempted by a real-time task at any moment, no Linux routine can safely be called from real-time tasks. However, some communication mechanism must be present. Simple FIFOs are used in RT-Linux for moving information between Linux processes or the Linux kernel and real-time processes. In a data-collecting application, for example, a real-time process would poll a device, and put the data into a FIFO. Linux processes can then be used for reading the data from the FIFO and storing it in the file, or displaying it on the screen. Currently, interrupts are disabled when a RT-FIFO is accessed. Since data are transmitted in small chunks, this does not compromise a low response time. Other approaches, notably using lock-free data structures, are also possible.

Modifications to the Linux kernel are primarily in three places:

The cli routine to disable interrupts is modified to simply clear a global variable controlling soft interrupt enable.

The sti routine to enable interrupts is modified to generate emulated interrupts for any pending soft interrupts.

The low-level "wrapper" routines that save and restore state around calls to handlers have been changed to use soft return from interrupt code instead of using the machine instruction.

FIGS. **2–6** are flow diagram depictions of the process of the present invention from which a person skilled in the art could form a suitable software routine for integrating a real-time processor with a general purpose operating system. Referring first to FIG. **2**, when an interrupt occurs **50**, control switches to a real-time handler. The handler does whatever needs to be done in the real-time executive **52**; i.e., further interrupts are disabled, the interrupt status is saved, and an acknowledgment (ACK) is issued. The ACK involves clearing the interrupt from the controller so that a later HARD_ENABLE will not trigger a second interrupt from the same signal. But the ACK must not permit the device to generate a second interrupt until it is hard-unmasked. Then the nature of the interrupt is determined **54**. If the soft interrupt enable flag is set **56**, then the stack is adjusted to fit the needs of the Linux handler, hard interrupts are re-enabled and control is passed, via a soft interrupt table, to the appropriate Linux "wrapper". The "wrapper" saves additional state and calls the Linux handler **58**—a program usually written in C language. When the handler returns control to the "wrapper" a soft return from interrupt is executed **60**. Soft return **60** from interrupt restores state and then checks to see if any other soft interrupts are pending. If not, a hard return from interrupt is executed **62** so that hard interrupts are re-enabled along a short path whereby real time interrupts can always be accepted. If there are interrupts pending **84** (FIG. **3**), then the highest priority one is processed.

Linux is reasonably easy to modify because, for the most part, the kernel code controls interrupt hardware through the routines cli() and sti(). In standard x86 Linux, these routines are actually assembly language macros that generate the x86 cli (clear interrupt bit) and sti (set interrupt bit) instructions for changing the processor control word.

As shown in FIG. **3**, interrupt handlers **58** in the RT-executive perform **66** whatever function is necessary for the selected RT system and then may pass **68** interrupts on to Linux. Since the real-time system is not involved in most I/O, most of the RT device interrupt handlers simply notify Linux. On the other hand, the timer interrupt increments timer variables, determines **72** whether a RT task needs to run, and passes interrupts **74** to Linux only at appropriate intervals.

If software interrupts are disabled, control simply returns **78** through interrupt return (iret). Other wise, control is passed **82** to the soft return from interrupt (S_IRET) **84**. Macro **85** (FIG. **4**) invokes the software handler corresponding to the interrupt that has the highest priority among pending and not masked ones.

The S_IRET code begins by saving minimal state and making sure that the kernel data address space is accessible. In the critical section surrounded by the actual cli and sti the software interrupt mask **82** is applied to the variable containing pending interrupts, and then looks **84** for the highest-priority pending interrupt. If there are no software interrupts to be processed, software interrupts are re-enabled **88** (FIG. **5**), the registers are restored, and the system returns from the interrupt. If there is an interrupt to process **76** (FIG. **6**), control is passed to its Linux "wrapper" **92**.

Each Linux "wrapper" has been modified to fix the stack so that it looks as if control has been passed directly from the hardware interrupt. This step is essential because Linux actually looks in the stack to see if the system was in user or kernel mode when the interrupt occurred. If Linux believes that the interrupt occurred in kernel mode, it will not call its own scheduler. The body of the wrapper has not been modified, but instead of terminating with an iret operation, the modified wrapper invokes S_IRET. Thus, wrappers essentially invoke each other until there are no pending interrupts left.

On re-enabling software interrupts, all pending ones, of course, should be processed. The code simulates a hardware interrupt. The flags and the return address are pushed onto the stack and S_IRET is used.

Individual disabling/enabling of interrupts is handled similarly.

Thus, Linux has been modified as little as possible in order to accommodate the real-time executive according to the present invention. The real-time executive approach might be used as a basis for significant redesign of Linux and similar operating systems. For example, device drivers often have real-time constraints. If the real-time requirements of the drivers were made explicit and moved into the RT-kernel, then configuration programs could attempt to find a feasible schedule rather than allowing users to find out by experiment whether device time constraints are feasible. It may also be possible to simplify design of the general purpose kernel by giving the emulation a cleaner semantics than the actual hardware.

The foregoing description of the invention has been presented for purposes of illustration and description and is not intended to be exhaustive or to limit the invention to the precise form disclosed, and obviously many modifications and variations are possible in light of the above teaching. The embodiments were chosen and described in order to best explain the principles of the invention and its practical application to thereby enable others skilled in the art to best utilize the invention in various embodiments and with various modifications as are suited to the particular use contemplated. It is intended that the scope of the invention be defined by the claims appended hereto.

What is claimed is:

1. A process for running a general purpose computer operating system using a real time operating system, including the steps of:

providing a real time operating system for running real time tasks and components and non-real time tasks;

providing a general purpose operating system as one of the non-real time tasks;

preempting the general purpose operating system as needed for the real time tasks; and

preventing the general purpose operating system from blocking preemption of the non-real time tasks.

2. A process according to claim **1**, further including the step of providing a software emulator to disable and enable interrupts from the general purpose operating system.

3. A process according to claim **2**, wherein the software emulator performs the steps of:

preventing the general purpose operating system from disabling hardware interrupts from hardware operating in real time; and

emulating hardware interrupt control to preserve interrupt behavior expected by the general purpose operating system with only minimal changes to the general purpose operating system code.

4. A process according to claim **3**, further including the steps of allocating the hardware interrupts to either real-time tasks and components of the real-time operating system or to the general purpose operating system.

5. A process according to claim **4**, wherein allocating the hardware interrupts further includes the steps of:

passing control directly to real-time tasks or components when the hardware generates interrupts that are allocated to real-time tasks or to components of the real-time operating system; and

passing control to the software emulator when the hardware generates interrupts that are allocated to the general purpose operating system.

6. A process according to claim **2**, further including the step of maintaining software interrupt control with the software emulator.

7. A process for running a general purpose computer operating system using a real time operating system, including the steps of:

providing a real time operating system for running real time tasks and components and non-real time tasks;

providing a general purpose operating system as one of the non-real time tasks;

preempting the general purpose operating system as needed for the real time tasks;

preventing the general purpose operating system from blocking preemption of the non-real time tasks;

providing a software emulator to disable and enable interrupts from the general purpose operating system;

marking interrupts as "soft disabled" and not "soft enabled" in response to requests from the general purpose operating system to disable interrupts;

marking interrupts as "pending" and returning control to an interrupted thread of execution in response to hardware interrupts allocated to the general purpose operating system if either the interrupted thread of execution was a real-time task or component of the real-time operating system or if the interrupt has been marked as "soft disabled";

emulating the interrupt in response to hardware interrupts allocated to the general purpose operating system if both the interrupted thread of execution was not a real-time task or component of the real-time operating system and the interrupt has been marked as "soft enabled"; and

marking interrupts as "soft enabled" and not "soft disabled" and then emulating any soft enabled interrupts in response to requests from the general purpose operating system to enable interrupts.

8. A process according to claim **7**, wherein the software emulator performs the steps of:

preventing the general purpose operating system from disabling hardware interrupts from hardware operating in real time; and

emulating hardware interrupt control to preserve interrupt behavior expected by the general purpose operating system with only minimal changes to the general purpose operating system code.

9. A process according to claim **8**, further including the steps of allocating the hardware interrupts to either real-time tasks and components of the real-time operating system or to the general purpose operating system.

10. A process according to claim **9**, wherein allocating the hardware interrupts further includes the steps of:

passing control directly to real-time tasks or components when the hardware generates interrupts that are allocated to real-time tasks or to components of the real-time operating system; and

passing control to the software emulator when the hardware generates interrupts that are allocated to the general purpose operating system.

11. A process according to claim **7**, where the step for emulating the interrupt further consists of the steps of:

marking the interrupt as "soft enabled" and not "soft disabled";

passing control to an appropriate interrupt handler of the general purpose operating system; and

restoring state after the interrupt handler of the general purpose operating system completes the non-real time task.

* * * * *

US005875341A

United States Patent [19]

Blank et al.

[11] Patent Number: 5,875,341

[45] Date of Patent: Feb. 23, 1999

[54] **METHOD FOR MANAGING INTERRUPT SIGNALS IN A REAL-TIME COMPUTER SYSTEM**

[75] Inventors: **Felix Blank; Peter Schicklinski; Bettina Sterr; Ursula Wiesinger**, all of Munich, Germany

[73] Assignee: **Siemens Aktiengesellshaft**, Munich, Germany

[21] Appl. No.: **719,413**

[22] Filed: **Sep. 24, 1996**

[30] **Foreign Application Priority Data**

Sep. 25, 1995 [DE] Germany 195 35 546.6

[51] **Int. Cl.**[6] **G06F 13/00**

[52] **U.S. Cl.** **395/733**; 395/735; 395/736; 395/737; 395/738; 395/739; 395/740; 395/741; 395/742; 395/591

[58] **Field of Search** 395/733, 735, 395/736, 737, 739, 565, 591, 734, 738, 740, 741, 742

[56] **References Cited**

U.S. PATENT DOCUMENTS

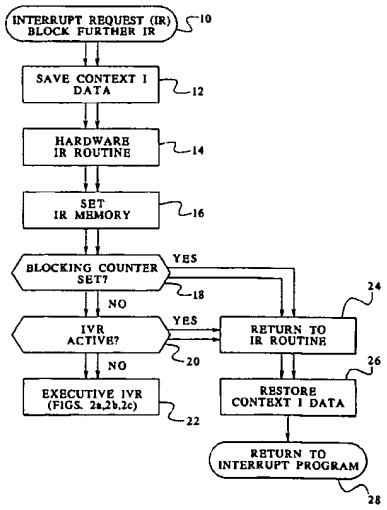
3,778,780	12/1973	Moore	340/172.5
3,905,025	9/1975	Davis et al.	340/172.5
4,015,243	3/1977	Kurpanek et al.	340/172.5
4,079,448	3/1978	Nguyen et al.	364/200
4,247,894	1/1981	Beismann et al.	364/200
4,488,227	12/1984	Miu et al.	364/200
4,719,565	1/1988	Moller	364/200
4,768,149	8/1988	Konopik et al.	364/200
4,885,681	12/1989	Umeno et al.	364/200
5,036,539	7/1991	Wrench, Jr. et al.	381/43
5,109,489	4/1992	Umeno et al.	395/275
5,247,671	9/1993	Adkins et al.	395/650
5,301,312	4/1994	Christopher, Jr. et al.	395/575
5,390,329	2/1995	Gaertner et al.	395/650
5,392,409	2/1995	Umeno et al.	395/400
5,396,632	3/1995	Gillet	395/725
5,418,968	5/1995	Gobeli	395/725
5,515,538	5/1996	Kleiman	395/733
5,535,380	7/1996	Bergkvist, Jr. et al.	395/550
5,539,729	7/1996	Bodnar	370/18
5,542,076	7/1996	Benson et al.	395/733
5,588,125	12/1996	Bennett	395/306
5,652,890	7/1997	Foster et al.	395/750
5,689,713	11/1997	Normoyle et al.	395/736
5,717,932	2/1998	Szczepanek et al.	395/733
5,717,933	2/1998	Mann	395/735

Primary Examiner—Meng-Ai T. An
Assistant Examiner—Valerie Darbe
Attorney, Agent, or Firm—Hill & Simpson

[57] **ABSTRACT**

A method for the operation of a computer system controlled by a real-time operating system, which computer system processes interrupt signals. Upon the occurrence of an interrupt signal, the computer system interrupts a program that is to be processed at that time. The acceptance of further interrupt signals is blocked, and an interrupt routine belonging to this interrupt signal is called. During the processing of this interrupt routine, a first part of the program parameters of the program that is interrupted upon the occurrence of the interrupt signal is intermediately stored, and at least one datum concerning the interrupt signal is stored in an interrupt memory. A branching takes place from the interrupt routine to an interrupt management routine (IVR), whereby the acceptance of further interrupt signals is again cleared during the processing of the IVR. During the processing of the IVR, the datum belonging to the interrupt signal is erased; the remaining part of the program parameters of the program that is interrupted upon the occurrence of the interrupt signal is intermediately stored. Dependent on the datum concerning the interrupt signal, at least one reaction routine belonging to this interrupt signal is activated. After the processing of the IVR, the operating system branches back to the program that was interrupted upon the occurrence of the interrupt signal, using the intermediately stored program parameters.

19 Claims, 4 Drawing Sheets



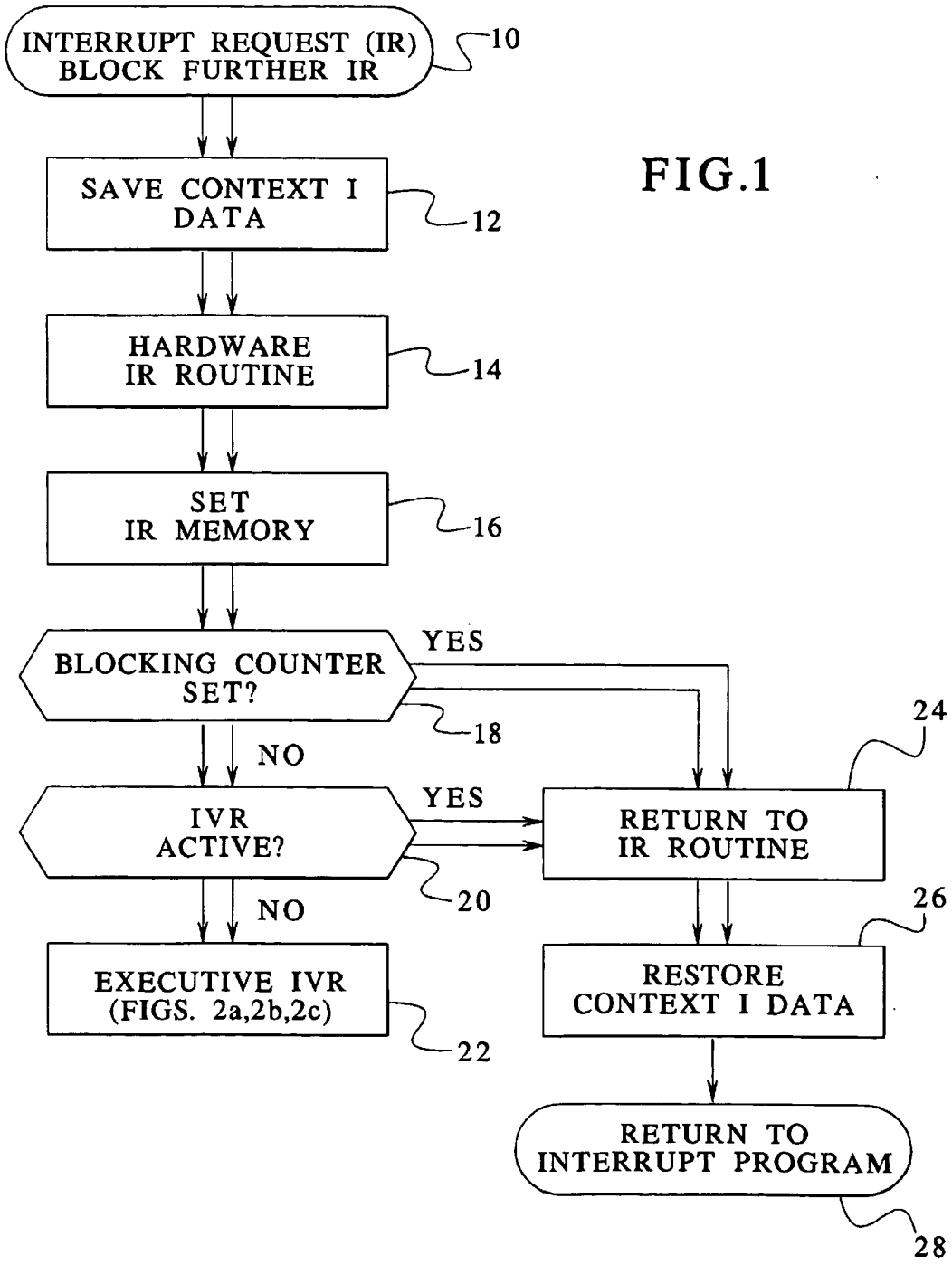

INTERRUPT REQUEST (IR)
BLOCK FURTHER IR
10
SAVE CONTEXT I DATA
12
HARDWARE IR ROUTINE
14
SET IR MEMORY
16
BLOCKING COUNTER SET?
YES
18
NO
IVR ACTIVE?
YES
20
NO
EXECUTIVE IVR (FIGS. 2a,2b,2c)
22
RETURN TO IR ROUTINE
24
RESTORE CONTEXT I DATA
26
RETURN TO INTERRUPT PROGRAM
28

FIG.1

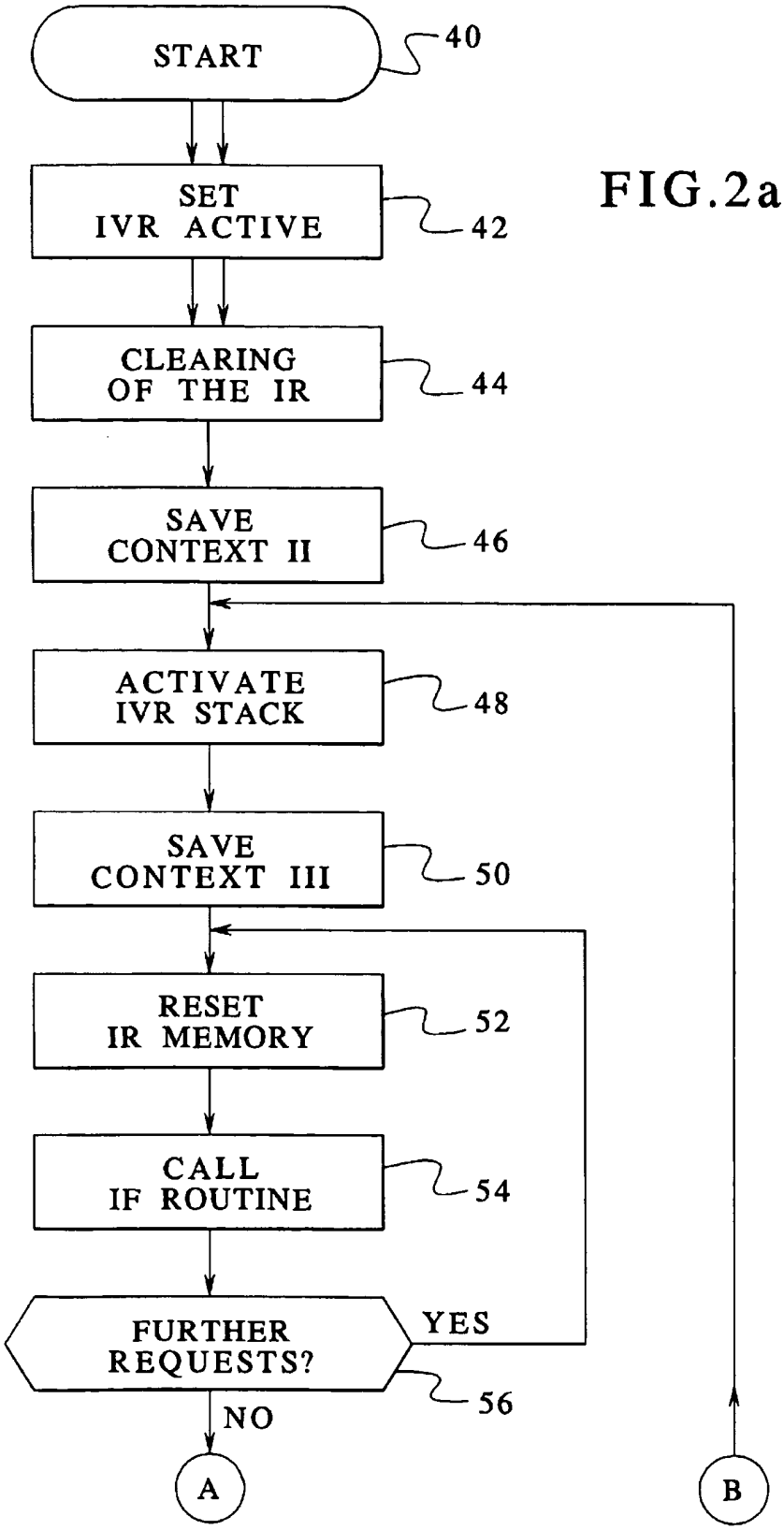

START
40
SET IVR ACTIVE
42
CLEARING OF THE IR
44
SAVE CONTEXT II
46
ACTIVATE IVR STACK
48
SAVE CONTEXT III
50
RESET IR MEMORY
52
CALL IF ROUTINE
54
FURTHER REQUESTS?
YES
56
NO
A
B

FIG.2a

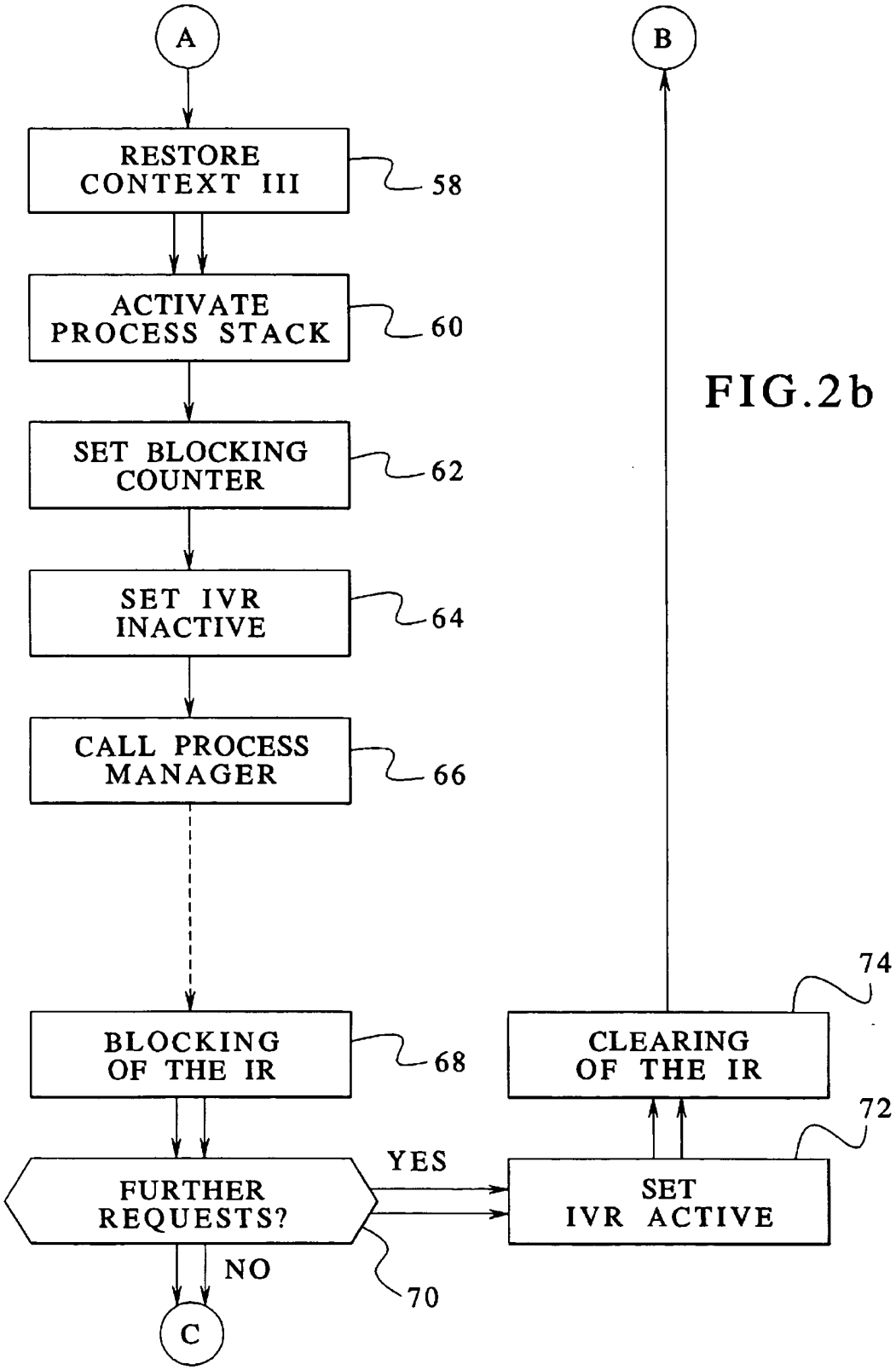
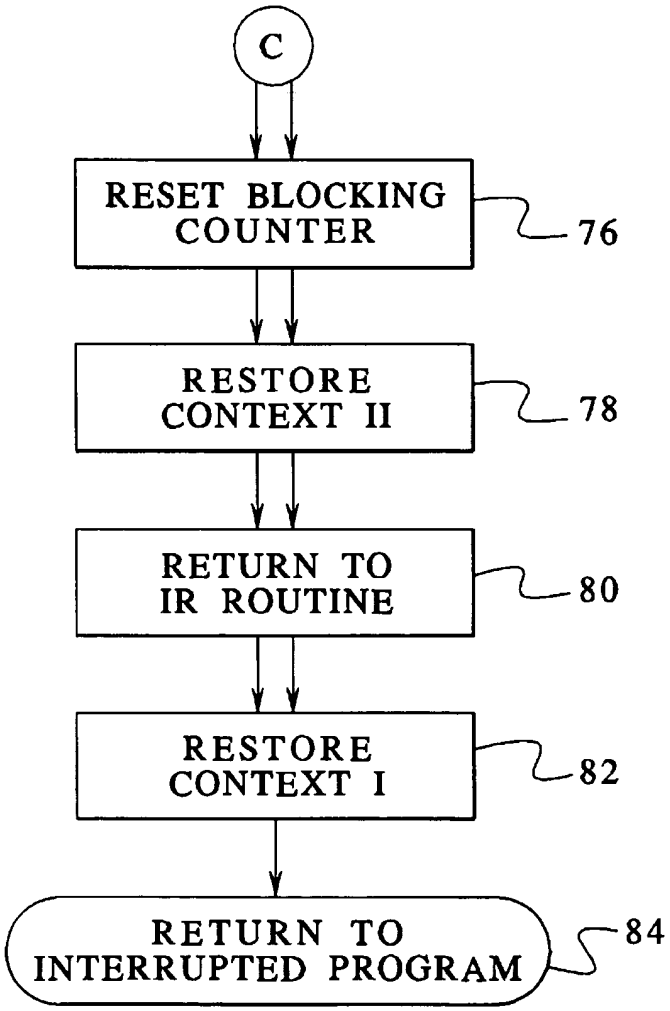

A
B
RESTORE CONTEXT III
58
ACTIVATE PROCESS STACK
60
SET BLOCKING COUNTER
62
SET IVR INACTIVE
64
CALL PROCESS MANAGER
66
BLOCKING OF THE IR
68
FURTHER REQUESTS?
YES
NO
70
C
CLEARING OF THE IR
74
SET IVR ACTIVE
72

FIG.2b

FIG.2c


C
RESET BLOCKING COUNTER
76
RESTORE CONTEXT II
78
RETURN TO IR ROUTINE
80
RESTORE CONTEXT I
82
RETURN TO INTERRUPTED PROGRAM
84

METHOD FOR MANAGING INTERRUPT SIGNALS IN A REAL-TIME COMPUTER SYSTEM

BACKGROUND OF THE INVENTION

The present invention relates to methods for operating a real-time computer system controlled by a real-time operating system, which computer system processes interrupt signals, e.g., of a technical process.

Methods of this type are used for example in real-time computer systems utilized in telecommunication modules of the public and private networks. A real-time computer system is characterized in that the processing of a program activated, for example, as a reaction to an interrupt signal can be interrupted within a known limited time interval, in order to continue the processing of this program only when higher-priority programs have been processed. During the processing of a program in telecommunication modules, e.g. up to about 50,000 interrupt signals can occur per second, which respectively require an interruption of a program to be processed at that time and a reaction to the interrupt signal.

Known real-time computer systems block the acceptance of further interrupt signals for a predetermined time interval, called the interrupt blocking time, so that interrupt signals coming in during this time interval are not taken into account is processed, and information or instructions can be lost. The interrupt blocking time is required so that the microprocessor can secure its registers, can react at least partly to the interrupt signal, and can again update its registers in order to be able to continue the interrupted program. The longer this interrupt blocking time is, all the more incoming interrupt signals go without being processed. An improved method for the operation of a real-time computer system, in which the interrupt blocking time is reduced, would be advantageous.

SUMMARY OF THE INVENTION

An object of the invention is to develop a real-time operating system component that permits the acquisition of, and reaction to, the highest possible number of occurring interrupt signals per time unit.

To that end, in an embodiment, the invention provides the following method steps: when an interrupt signal occurs, the real-time computer system interrupts a program processing at that time; the acceptance of further interrupt signals is blocked, and an interrupt routine belonging to this interrupt signal is called; during the processing of this interrupt routine, a first group of one or more program parameters of the program that was interrupted upon the occurrence of the interrupt signal is intermediately stored; at least one datum concerning the interrupt signal is stored in an interrupt memory; a branching takes place from the interrupt routine to an interrupt management routine, whereby the acceptance of further interrupt signals is again cleared during the processing of the interrupt management routine; during the processing of the interrupt management routine, the datum belonging to the interrupt signal in the interrupt memory is erased; the remainder of the program parameters of the program that was interrupted upon the occurrence of the interrupt signal are intermediately stored; depending on the datum concerning the interrupt signal in the interrupt memory, at least one reaction routine belonging to this interrupt signal is activated and, if warranted, is processed with activation of the real-time operating system; and after the processing of the interrupt management routine, the real-time operating system branches back to the program

that was interrupted upon the occurrence of the interrupt signal, using the intermediately stored program parameters.

The invention is based on the consideration that a shortening of the interrupt blocking time contributes to the capacity for processing the greatest possible number of interrupt signals. The interrupt blocking time can be shortened when previously necessary command sequences from an interrupt routine are transferred to external storage, which sequences fall in the interrupt blocking time in known methods. An interrupt management routine can be provided for reaction routines belonging to the individual interrupt routines, in which those actions for the processing of the interrupt signal that do not absolutely have to be executed immediately upon the occurrence of the interrupt signal are respectively transferred to external storage from the associated interrupt routines. These actions include e.g. the securing of some of the register data of the microprocessor, the majority of the steps previously carried out in the interrupt routine for the reaction to the interrupt signal, and the restoring of the register data of the microprocessor that did not have to be secured immediately upon the occurrence of the interrupt signal.

The register data of the microprocessor that must be secured immediately upon the occurrence of the interrupt signal is referred to in the following description of the first part of the program parameters (context I parameters) of the program that is interrupted upon the occurrence of the interrupt signal, and can be, e.g. the content of the flag register of the microprocessor. The rest of the data to be secured are referred to as the remaining part of the program parameters (context II and III parameters) of the program that is interrupted upon the occurrence of the interrupt signal.

Since during or upon the processing of the interrupt management routine the acceptance of further interrupt signals is again cleared, the interrupt blocking time is shortened considerably in contrast to known methods. Consequently, it is also the case that fewer interrupt signals are thereby lost, and fewer data that are available only for a short time upon occurrence of the interrupt signal are lost.

An advantageous construction of the inventive method is that the clearing of the acceptance of further interrupt signals ensues at the beginning of the interrupt management routine. Due to a clearing of the acceptance of further interrupt signals that takes place as early as possible, the interrupt blocking time is determined only through an absolutely necessary time interval, and is thus maximally shortened.

In a preferred embodiment of the invention, the storing of the remainder part of the program parameters of the program that is interrupted upon the occurrence of the interrupt signal is intermediately stored, partly in a first stack memory for program parameters (context II), which memory is allocated to the interrupted program, and partly in a second stack memory for program parameters (context III), which memory is allocated to the interrupt management routine. In this way, memory space can be saved, since the program parameters stored in the second stack memory do not have to be additionally stored for each interrupted program.

These and other features of the invention are discussed in greater detail below in the following detailed description of the presently preferred embodiments with reference to the accompanying drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is a flow chart of method steps undertaken upon the occurrence of an interrupt signal.

FIGS. 2*a*, 2*b* and 2*c* constitute a flow chart of the method steps executed in the context of the processing of an interrupt management routine.

DETAILED DESCRIPTION OF THE PRESENTLY PREFERRED EMBODIMENTS

In the figures, the method steps in which the acceptance of interrupt signals is blocked are indicated by double arrows. On the other hand, single arrows identify method steps in which the acceptance of further interrupt signals is cleared.

FIG. 1 illustrates a flow chart of steps of a method embodying principles of the invention which is undertaken upon the occurrence of an interrupt signal (step **10**). Immediately after the occurrence of the interrupt signal, the acceptance of further interrupt signals is blocked by the real-time computer system. So that the microprocessor can continue the processing of the interrupted program at a later time after the interruption by the interrupt signal, the state of its registers must be stored before further steps for the processing of the current interrupt signal are carried out. As already stated, that part of the program parameters of the program interrupted upon the occurrence of the interrupt signal that must be stored or saved immediately after the occurrence of the interrupt is designated context I parameters (even though it may be a single item). This saving ensues while an interrupt block (step **12**) is in effect, whereby the program parameters are stored in a stack memory in a predetermined sequence.

After the microprocessor is again allowed to alter the register contents after the saving of the register data, it can confirm the acceptance of the interrupt signal to other components of the real-time computer system, such as e.g. circuits for accepting interrupt (step **14**). In known methods, the method steps **12** and **14** are carried out within the interrupt routine, and thus should also be regarded here as components of an interrupt routine.

So that the interrupt signal can be processed later, the microprocessor stores a datum that points to the interrupt signal in an interrupt memory (step **16**). This interrupt memory can be e.g. a memory cell comprising several bits combined into one memory word, in which a particular bit is set according to the type of interrupt signal that occurs, so that it can be determined later which interrupt signal occurred. This possibility is described below.

The microprocessor subsequently checks whether, at the time of the occurrence of the interrupt signal, program routines are being processed in the real-time computer system that are time-critical and thus may not be interrupted (step **18**). The number of such non-interruptible programs is stored by the microprocessor in a memory cell, e.g. in a blocking (or block) counter with a determined initial value. The blocking counter is increased upon each calling of a non-interruptible program, and is correspondingly decreased after the processing of a non-interruptible program. In method step **18**, the microprocessor can also determine whether non-interruptible programs are being processed at the current time. If this is not the case, i.e. if the memory cell of the blocking counter has the determined initial value, the microprocessor checks whether an interrupt management routine (IVR) is already active, by querying a state memory cell in which the state of the interrupt management routine is stored (step **20**). If the interrupt management routine is not activated, it is called (step **22**). The method steps to be executed within the interrupt management routine are explained further below on the basis of FIGS. 2*a*, 2*b* and 2*c*.

If it is determined in method step **18** that the content of the blocking counter deviates from its initial value, i.e. a non-interruptible program routine is active, or if it is determined in method step **20** on the basis of the state memory cell that the interrupt management routine has already been activated, a jump back to the interrupt routine ensues (step **24**). In the interrupt routine, context I is restored, by reading the data for the processor registers out of the stack memory again, in a sequence that is the reverse of that in which they were stored (step **26**). Afterwards, there ensues the clearing of the acceptance of further interrupt signals, and a return takes place to the program that was interrupted upon the occurrence of the interrupt, in order to process it further (step **28**).

Up to the last method step **28**, all method steps in FIG. **1** are carried out while the interrupt block is in effect, so that up to this point there results no essential advantage over the prior art with respect to the interrupt blocking time. The savings is first visible on the basis of the method steps according to FIGS. 2*a*, 2*b*, 2*c*, since in the method sequence shown there the majority of the method steps are carried out with the clearing of the acceptance of further interrupt signals.

FIGS. 2*a*, 2*b*, and 2*c* show a flow chart of steps of an interrupt management routine. After the interrupt management routine has been called in method step **22**, the microprocessor begins the processing of the method steps of the interrupt management routine (step **40**).

Subsequently, the state memory cell is set (step **42**). Upon the occurrence of further interrupt signals, it can thereby be determined in method step **20** whether the interrupt management routine has already been activated, since in a correct execution of the method it may be activated only once.

In order to shorten the interrupt blocking time over known methods to the greatest possible extent, the acceptance of further interrupt signals is cleared at once (step **44**). After method step **44** has been processed, it is thus possible to accept and process further interrupt signals. These further interrupt signals are registered in the interrupt memory by the microprocessor as data in the form of set bits, in method step **16**. Upon the subsequent continuation of the processing of the method steps in the interrupt management routine, these further interrupt signals can be taken into account.

The program parameters of the program that was interrupted upon the occurrence of the interrupt signal, which parameters were not yet stored as context I in method step **12**, are divided into two groups. One group is formed by program parameters that are intermediately saved or, respectively, intermediately stored in a stack memory for program parameters, which memory is allocated to the interrupted program; this group of program parameters is designated or referred to herein as context II parameters. The further group of remaining program parameters is intermediately stored in a stack memory for program parameters, allocated to the interrupt management routine, and is designated or referred to as context III parameters.

By means of this division of the remaining program parameters, a considerable savings in memory space can be achieved, since the context III parameter does not have to be respectively stored in the stack memory of the interrupted program, but rather only in the stack memory of the interrupt management routine.

The saving of the context II parameters (step **46**) ensues, as mentioned, in a stack memory allocated to the interrupted program. The stack memory of the interrupt management

routine is subsequently activated (step **48**), and a saving of context III parameters ensues in this stack memory (step **50**). A later continuation of the program that was interrupted upon the occurrence of the interrupt signal is possible by means of the storing of the program parameters. The resetting of the bit allocated to the interrupt signal in the interrupt memory, which was set upon the occurrence of the interrupt in method step **16**, subsequently ensues in step **52**. It is thereby possible that the same interrupt signal can be again accepted and evaluated that is being processed at that time by the interrupt management routine.

After all preparations for the calling of a routine for the processing of the interrupt signal are concluded, the processing routine corresponding to the interrupt signal is called (step **54**). During the execution of the processing routine, further interrupt signals can be accepted, which are then noted by the setting of corresponding bits in the interrupt memory in method step **16**. After the execution of the processing routine in method step **54**, the interrupt memory is queried for set bits by the microprocessor (step **56**). If bits of this sort are set in the interrupt memory, the method is continued in method step **52**. The method thereby finds itself in an operational loop formed by method steps **52** to **56**. During the processing of the operational loop, further interrupt signals can be accepted that are signaled via the interrupt memory of the interrupt management routine, and are taken into account during the query in method step **56**. Since, in contrast to known methods, the processing of the interrupt signals ensues with interrupt clearance, the interrupt blocking time can be considerably shortened.

It is advantageous if the query for set bits in the interrupt memory in method step **56** takes into account the priorities of the accepted interrupt signals. In this way, the individual bits of the interrupt memory can be queried corresponding to an order of priority in the memory word of the interrupt memory, e.g. respectively from right to left, whereby a priority of the individual bits is determined that decreases from right to left in the memory word.

The continuation of the method with the method steps of FIG. **2***b* does not ensue until all set bits in the interrupt memory have been reset (cf. method step **52**) and the corresponding processing routines have been executed (cf. method step **54**). The restoring of context III parameters (step **58**) ensues by reading the program parameters of context III parameters out from the stack memory allocated to the interrupt management routine in the reverse sequence from that in which they were stored.

There are two different possibilities for the calling of the interrupt management routine. The first possibility, already described, is the calling according to method step **22**. In this case, the method steps **24** to **28** are not processed. As a second possibility, the interrupt management routine is called via the real-time operating system, if non-interruptible programs are being processed at the time of the occurrence of the interrupt signal, and the blocking counter was thereby already set in method step **18**. The method steps **24** to **28** would accordingly have already been executed before the calling of the interrupt management routine by the real-time operating system, as explained above, and thus may not be executed again within the interrupt management routine.

The calling of the interrupt management routine via the real-time operating system can ensue in such a way that after the execution of a non-interruptible program and after the decrease in the blocking counter it is checked as to whether the blocking counter has the initial value, and whether at least one bit allocated to an interrupt signal is set in the interrupt memory. If both conditions are met, the interrupt management routine is called.

In order to distinguish the two possibilities and to be able to process them correspondingly, a program stack memory allocated to a program management routine can be activated after the restoring of context III parameters in method step **58**, by loading the stack register of the microprocessor with the address of the stack memory of the interrupted program (step **60**). The program management routine is a component of the real-time operating system, and coordinates the processing of programs by noting programs to be processed and starting their processing in a predetermined sequence.

An increase in the blocking counter subsequently ensues (step **62**), since the following method steps **64** and **66** may not be interrupted. Subsequently, the interrupt management routine is marked as inactive, e.g. by means of a setting of the state memory cell to zero (step **64**), in order to enable the interrupt management routine to be processed again.

A calling of the program management routine (step **66**) then ensues, which if warranted brings another program to execution. After the termination thereof, the program management routine causes a return to the interrupt management routine and a blocking of the acceptance of further interrupt signals (step **68**).

Since during the processing of method steps **56** to **68** further interrupt signals could have been accepted, which signals have as a consequence the setting of allocated bits in method step **16** in the interrupt memory, during the processing of the interrupt management routine, before the termination thereof, it is again queried whether bits are still set in the interrupt memory (step **70**). If this is the case, the interrupt management routine is not terminated, but rather is set active again using the corresponding state memory cell (step **72**), and the acceptance of further interrupt signals is again cleared (step **74**). The method is subsequently continued in method step **48**. The interrupt management routine is now in a further operational loop consisting of method steps **48** to **74**. This further operational loop can be terminated only in method step **70**, if all bits in the interrupt memory have already been reset. In this case, there ensues a resetting of the blocking counter via a corresponding lowering of its value (step **76**).

There subsequently ensues a restoring of context II (step **78**), in that a switchover takes place to the stack memory of the program that was interrupted upon the occurrence of the interrupt signal, and the program parameters stored there are read again in a sequence that is the reverse of that in which they were stored.

The return to the interrupt routine ensues in step **80**, in which context I is then restored (step **82**). Subsequently, a return takes place to the program that was interrupted upon the occurrence of the interrupt signal (step **84**). The interrupt management routine is thereby concluded.

Although modifications and changes may be suggested by those skilled in the art, it is the intention of the inventors to embody within the patent warranted hereon all changes and modifications as reasonably and properly come within the scope of their contribution to the art.

We claim:

1. A method for operating a real-time computer system controlled by a real-time operating system, which computer system processes interrupt signals, comprising the following steps:

when an interrupt signal occurs, the real-time computer system interrupts any program processing at that time;

the acceptance of further interrupt signals is blocked, and an interrupt routine for this interrupt signal is executed;

during the processing of the interrupt routine, a first group of program parameters of the program interrupted upon the occurrence of the interrupt signal is intermediately stored;

at least one datum concerning the interrupt signal is stored in an interrupt memory, branching to an interrupt management routine depending upon whether at least one program routine, which may not be interrupted, is being processed in the computer system;

whereby the acceptance of further interrupt signals is again cleared during the processing of the interrupt management routine;

during the processing of the interrupt management routine, the datum belonging to the interrupt signal in the interrupt memory is erased,

the remainder of the program parameters of the program that was interrupted upon an occurrence of the interrupt signal are intermediately stored;

depending on the datum of the interrupt signal in the interrupt memory, at least one reaction routine belonging to this interrupt signal is activated and, if warranted, is processed with activation of the real-time operating system;

after the processing of the interrupt management routine, the real-time operating system branches back to the program that was interrupted upon the occurrence of the interrupt signal, using the intermediately stored program parameters.

2. The method according to claim **1**, characterized in that the clearing of the acceptance of further interrupt signals ensues at the beginning of the interrupt management routine.

3. The method according to any of claims **1** and **2**, characterized in that the first group of the program parameters comprises the content of processor registers at the time of the interruption of a microprocessor controlled by the real-time operating system.

4. The method according to any of claims **1** and **2**, characterized in that the remainder of the program parameters that remains during the processing of the interrupt management routine contains information concerning the state of the program that is halted upon the occurrence of the interrupt signal, and information that must be restored for the later continuation of the interrupted program.

5. The method according to claim **1**, characterized in that the storing of the remainder of the program parameters of the program that is interrupted upon the occurrence of the interrupt signal are intermediately stored partly in a first stack memory for program parameters, which memory is allocated to the interrupted program, and partly in a second stack memory for program parameters, which memory is allocated to the interrupt management routine.

6. The method according to claim **1**, characterized in that given several interrupt signals that follow upon one another, data belonging to these interrupt signals are stored in the interrupt memory.

7. The method according to claim **6**, characterized in that the data concerning accepted interrupt signals are stored according to a predetermined order of priority, and in that the processing of the interrupt signals ensues by means of the interrupt management routine, dependent on this order of priority.

8. The method according to claim **1**, characterized in that during the processing of the interrupt management routine, further data concerning interrupt signals are stored in the interrupt memory, which data occur during the processing of the interrupt management routine, and in that the interrupt management routine is not terminated until no data concerning interrupt signals are stored in the interrupt memory.

9. A method for operating a real-time computer system controlled by a real-time operating system, which computer system processes interrupt signals, comprising the following steps:

when an interrupt signal occurs, the real-time computer system interrupts any program processing at that time;

the acceptance of further interrupt signals is blocked, and an interrupt routine for this interrupt signal is executed,

during the processing of the interrupt routine, a first group of program parameters of the program interrupted upon the occurrence of the interrupt signal is intermediately stored;

at least one datum concerning the interrupt signal is stored in an interrupt memory, branching to the interrupt routine ensuing dependent on the value of a blocking counter that counts the calls of non-interruptible routines of the real-time operating system;

the interrupt routine is temporarily halted and an interrupt management routine is executed, whereby the acceptance of further interrupt signals is again cleared during the processing of the interrupt management routine,

the clearing of the acceptance of further interrupt signals ensuing at the beginning of the interrupt management routine;

during the processing of the interrupt management routine, the datum belonging to the interrupt signal in the interrupt memory is erased;

the remainder of the program parameters of the program that was interrupted upon an occurrence of the interrupt signal are intermediately stored;

depending on the datum of the interrupt signal in the interrupt memory, at least one reaction routine belonging to this interrupt signal is activated and, if warranted, is processed with activation of the real-time operating system; and

after the processing of the interrupt management routine, the real-time operating system branches back to the program that was interrupted upon the occurrence of the interrupt signal, using the intermediately stored program parameters.

10. The method according to claim **9**, characterized in that upon each calling of a non-interruptible routine the value of the blocking counter is increased, and after the processing of a non-interruptible routine the value of the blocking counter is correspondingly lowered.

11. The method according to claim **9** or **10**, characterized in that, using the intermediately stored program parameters, before the interrupt routine is processed the real-time operating system branches back to the program that was interrupted upon the occurrence of the interrupt signal, if the blocking counter does not have a predetermined initial value or the interrupt management routine has not yet been fully processed.

12. The method according to one of claims **9** or **10**, characterized in that the branching to the interrupt management routine ensues when the blocking counter has the predetermined initial value, and at least one datum concerning an interrupt signal is stored in the interrupt memory.

13. The method according to one of claims **5** to **8** or **10**, characterized in that a restoring of the program parameters intermediately stored on the second stack memory allocated

to the interrupt management routine does not ensue until no datum is stored in the interrupt memory.

14. The method according to one of claims **5** to **8** or **10**, characterized in that a restoring of the program parameters intermediately stored on the first stack memory allocated to the interrupted program does not ensue until no datum is stored in the interrupt memory.

15. A method for operating a real-time computer system controlled by a real-time operating system, which computer system processes interrupt signals, comprising the following steps:

when an interrupt signal occurs, the real-time computer system interrupts any program processing at that time;

the acceptance of further interrupt signals is blocked, and an interrupt routine for this interrupt signal is executed;

during the processing of the interrupt routine, a first group of program parameters of the program interrupted upon the occurrence of the interrupt signal is intermediately stored;

at least one datum concerning the interrupt signal is stored in an interrupt memory, branching to an interrupt management routine depending upon whether at least one program routine, which may not be interrupted, is being processed in the computer system;

whereby the acceptance of further interrupt signals is again cleared during the processing of the interrupt management routine,

the clearing of the acceptance of further interrupt signals ensuing at the beginning of the interrupt management routine;

during the processing of the interrupt management routine, the datum belonging to the interrupt signal in the interrupt memory is erased;

the remainder of the program parameters of the program that was interrupted upon an occurrence of the interrupt signal are intermediately stored;

depending on the datum of the interrupt signal in the interrupt memory, at least one reaction routine belonging to this interrupt signal is activated and, if warranted, is processed with activation of the real-time operating system;

after the processing of the interrupt management routine, the real-time operating system branches back to the program that was interrupted upon the occurrence of the interrupt signal, using the intermediately stored program parameters; and

given several interrupt signals that follow upon one another, data belonging to these interrupt signals are stored in the interrupt memory.

16. A method for operating a real-time computer system controlled by a real-time operating system, which computer system processes interrupt signals, comprising the following steps:

when an interrupt signal occurs, the real-time computer system interrupts any program processing at that time;

the acceptance of further interrupt signals is blocked, and an interrupt routine for this interrupt signal is executed;

during the processing of the interrupt routine, a first group of program parameters of the program interrupted upon the occurrence of the interrupt signal is intermediately stored;

at least one datum concerning the interrupt signal is stored in an interrupt memory, branching to an interrupt management routine depending upon whether at least one program routine, which may not be interrupted, is being processed in the computer system;

whereby the acceptance of further interrupt signals is again cleared during the processing of the interrupt management routine,

the clearing of the acceptance of further interrupt signals ensuing at the beginning of the interrupt management routine;

during the processing of the interrupt management routine, the datum belonging to the interrupt signal in the interrupt memory is erased;

the remainder of the program parameters of the program that was interrupted upon an occurrence of the interrupt signal are intermediately stored;

depending on the datum of the interrupt signal in the interrupt memory, at least one reaction routine belonging to this interrupt signal is activated and, if warranted, is processed with activation of the real-time operating system, the interrupt management routine not being terminated until no datum concerning an interrupt signal is stored in the interrupt memory;

during the processing of the interrupt management routine, further data concerning interrupt signals are stored in the interrupt memory, which data occur during the processing of the interrupt management routine; and

after the processing of the interrupt management routine, the real-time operating system branches back to the program that was interrupted upon the occurrence of the interrupt signal, using the intermediately stored program parameters.

17. A method for operating a real-time computer system controlled by a real-time operating system, which computer system processes interrupt signals, comprising the following steps:

when an interrupt signal occurs, the real-time computer system interrupts any program processing at that time;

the acceptance of further interrupt signals is blocked, and an interrupt routine for this interrupt signal is executed;

during the processing of the interrupt routine, a first group of program parameters of the program interrupted upon the occurrence of the interrupt signal is intermediately stored;

at least one datum concerning the interrupt signal is stored in an interrupt memory, branching to an interrupt management routine depending upon whether at least one program routine, which may not be interrupted, is being processed in the computer system;

whereby the acceptance of further interrupt signals is again cleared during the processing of the interrupt management routine,

the clearing of the acceptance of further interrupt signals ensuing at the beginning of the interrupt management routine;

during the processing of the interrupt management routine, the datum belonging to the interrupt signal in the interrupt memory is erased;

the remainder of the program parameters of the program that was interrupted upon an occurrence of the interrupt signal are intermediately stored, the remainder of the program parameters of the program that is interrupted upon the occurrence of the interrupt signal being intermediately stored partly in a first stack memory for program parameters, which memory is allocated to the

interrupted program, and partly in a second stack memory for program parameters, which memory is allocated to the interrupt management routine;

depending on the datum of the interrupt signal in the interrupt memory, at least one reaction routine belonging to this interrupt signal is activated and, if warranted, is processed with activation of the real-time operating system; and

after the processing of the interrupt management routine, the real-time operating system branches back to the program that was interrupted upon the occurrence of the interrupt signal, using the intermediately stored program parameters.

18. The method according to claim **17**, characterized in that a restoring of the program parameters intermediately stored on the second stack memory allocated to the interrupt management routine does not ensue until no datum is stored in the interrupt memory.

19. The method according to claim **17**, characterized in that a restoring of the program parameters intermediately stored on the first stack memory allocated to the interrupted program does not ensue until no datum is stored in the interrupt memory.

* * * * *

US006499078B1

(12) **United States Patent**
Beckert et al.

(10) **Patent No.: US 6,499,078 B1**
(45) **Date of Patent: Dec. 24, 2002**

(54) **INTERRUPT HANDLER WITH PRIORITIZED INTERRUPT VECTOR GENERATOR**

(75) Inventors: **Richard D. Beckert**, Lake Stevens, WA (US); **Mark M. Moeller**, Bellingham, WA (US); **Patrick Mullarky**, Bellevue, WA (US)

(73) Assignee: **Microsoft Corporation**, Redmond, WA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **09/357,064**

(22) Filed: **Jul. 19, 1999**

(51) **Int. Cl.**[7] **G06F 9/48**; G06F 13/26
(52) **U.S. Cl.** **710/260**; 710/262; 710/266
(58) **Field of Search** 710/260, 264, 710/266, 269, 1, 48, 49, 52, 262; 701/1; 700/1

(56) **References Cited**

U.S. PATENT DOCUMENTS

4,734,882 A * 3/1988 Romagosa
5,225,974 A * 7/1993 Mathews et al.
5,487,002 A * 1/1996 Diller et al.
5,594,905 A 1/1997 Mital 395/733
5,894,578 A * 4/1999 Qureshi et al.
5,923,887 A * 7/1999 Dutton

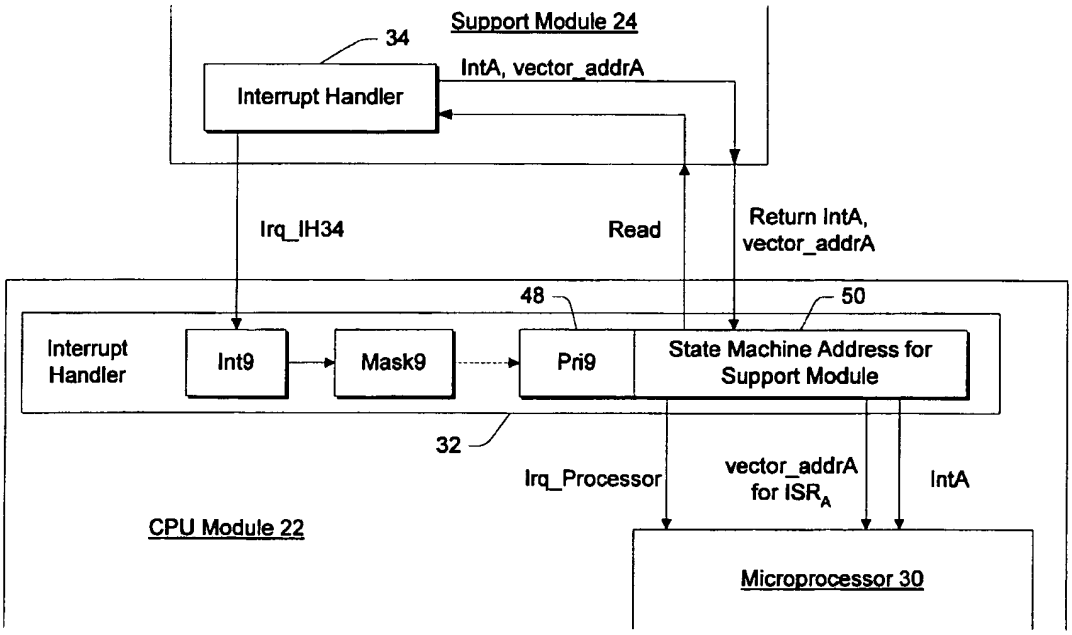
* cited by examiner

Primary Examiner—Gopal C. Ray
(74) *Attorney, Agent, or Firm*—Lee & Hayes, PLLC

(57) **ABSTRACT**

A hardware-implemented interrupt handler external to a processor handles interrupts destined for the processor. The interrupt handler has a programmable prioritized interrupt array with programmable registers that identify priority levels and handling processes for handling one or more interrupts. The interrupt handler also has an interrupt scanning state machine that scans the prioritized interrupt following receipt of an interrupt to extract the priority level and handling process associated with the interrupt. The interrupt handler is designed to handle interrupts in significantly less time than software implementations, thereby making the handler favorable for real time systems.

28 Claims, 4 Drawing Sheets



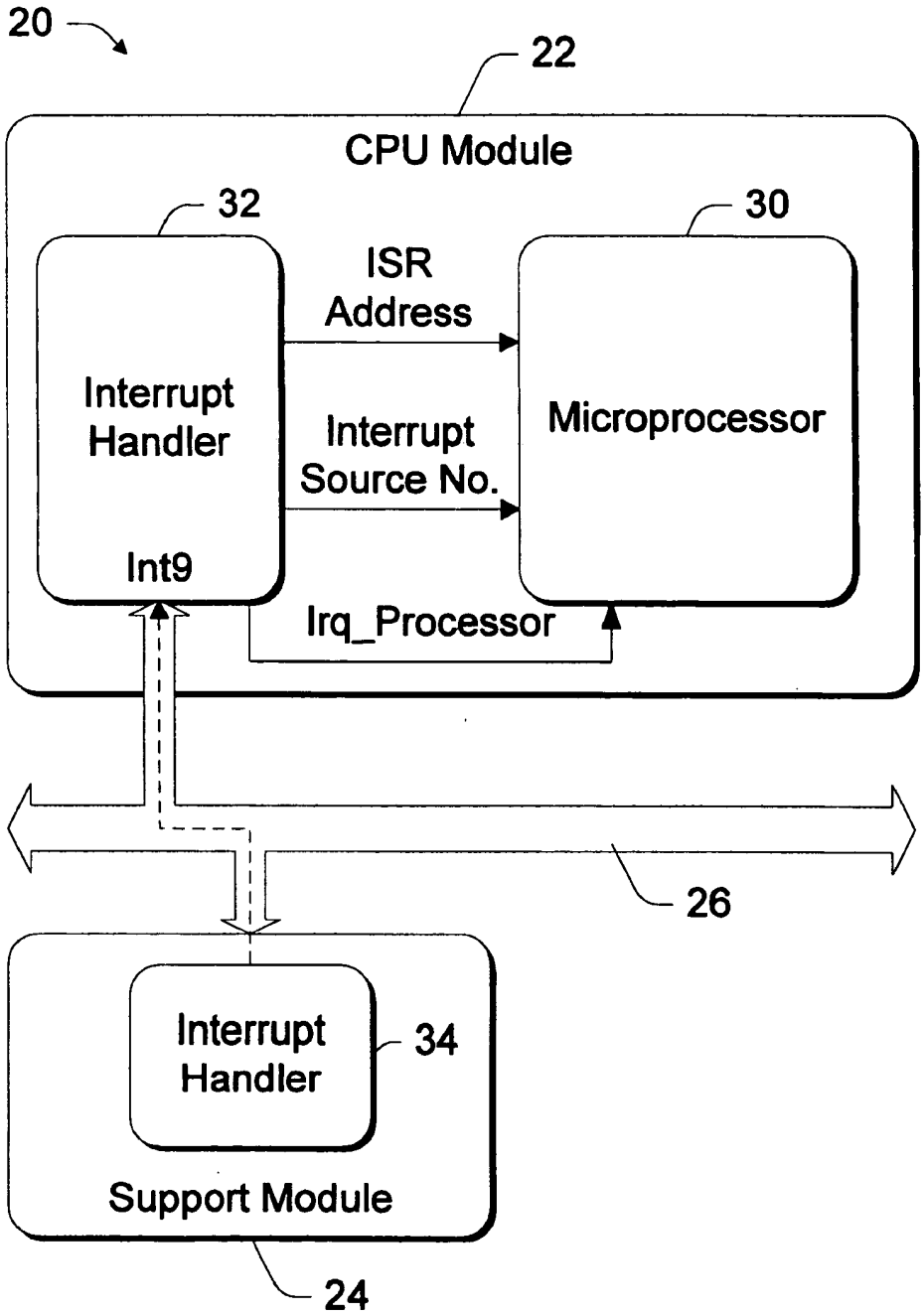

20
22
CPU Module
32
30
ISR
Address
Interrupt
Handler
Interrupt
Source No.
Microprocessor
Int9
Irq_Processor
26
Interrupt
Handler
34
Support Module
24

Fig. 1

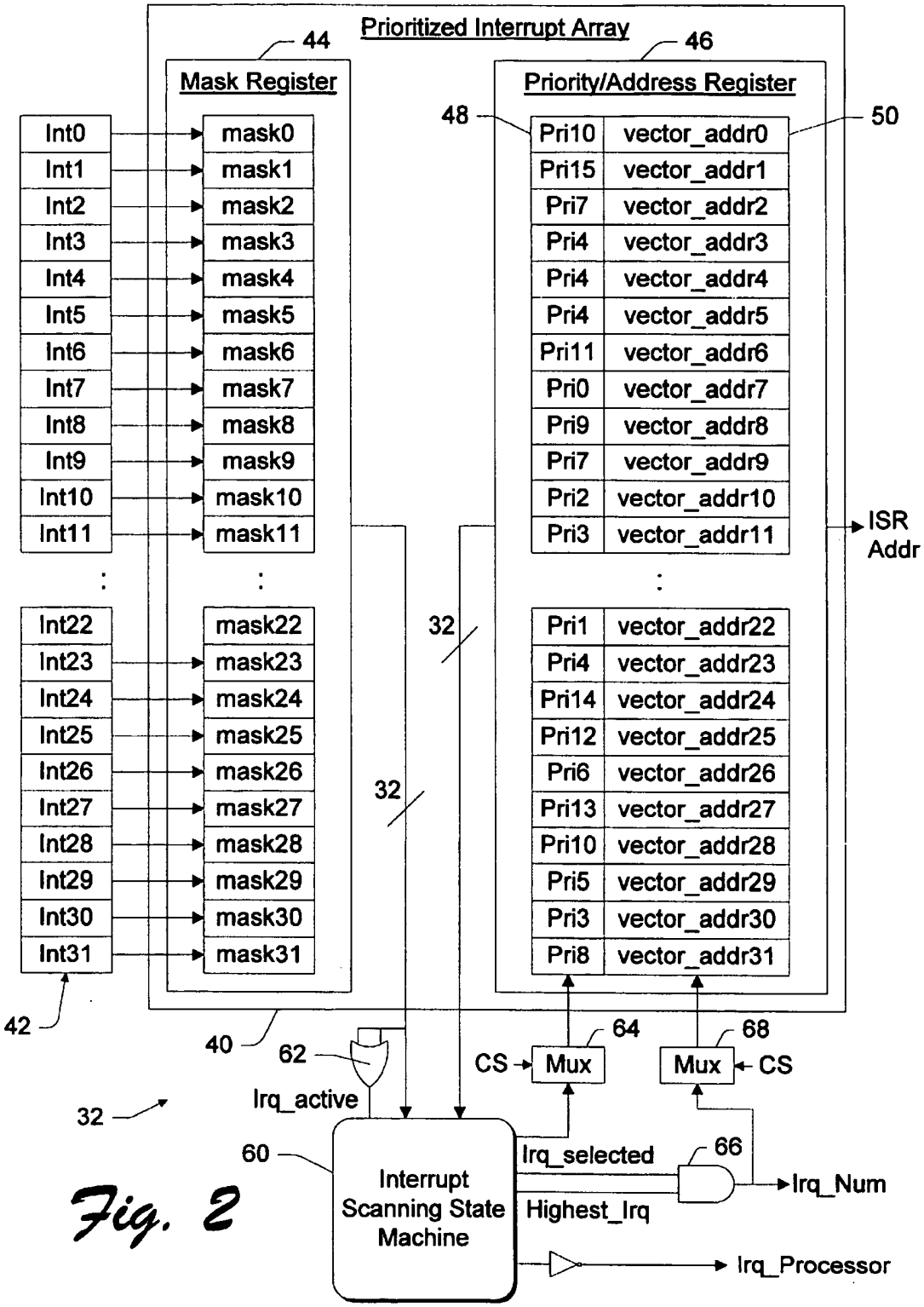

Prioritized Interrupt Array
44
Mask Register
46
Priority/Address Register
48
50
Int0
Int1
Int2
Int3
Int4
Int5
Int6
Int7
Int8
Int9
Int10
Int11
Int22
Int23
Int24
Int25
Int26
Int27
Int28
Int29
Int30
Int31
mask0
mask1
mask2
mask3
mask4
mask5
mask6
mask7
mask8
mask9
mask10
mask11
mask22
mask23
mask24
mask25
mask26
mask27
mask28
mask29
mask30
mask31
Pri10 vector_addr0
Pri15 vector_addr1
Pri7 vector_addr2
Pri4 vector_addr3
Pri4 vector_addr4
Pri4 vector_addr5
Pri11 vector_addr6
Pri0 vector_addr7
Pri9 vector_addr8
Pri7 vector_addr9
Pri2 vector_addr10
Pri3 vector_addr11
Pri1 vector_addr22
Pri4 vector_addr23
Pri14 vector_addr24
Pri12 vector_addr25
Pri6 vector_addr26
Pri13 vector_addr27
Pri10 vector_addr28
Pri5 vector_addr29
Pri3 vector_addr30
Pri8 vector_addr31
ISR Addr
32
32
42
40
62
Irq_active
32
64
68
CS
Mux
Mux
CS
60
Interrupt Scanning State Machine
Irq_selected
66
Irq_Num
Highest_Irq
Irq_Processor

Fig. 2

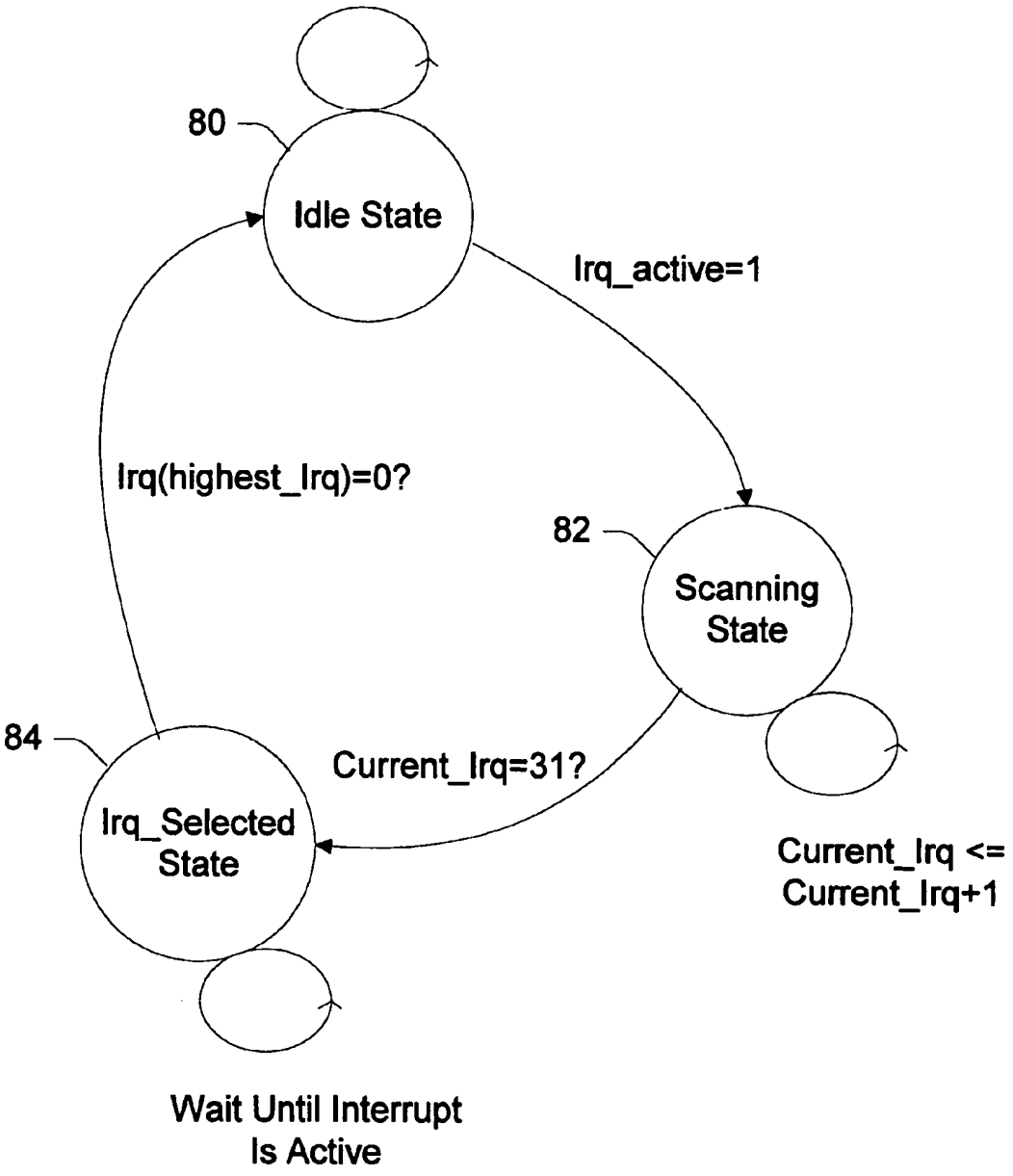

80
Idle State
Irq_active=1
Irq(highest_Irq)=0?
82
Scanning State
84
Current_Irq=31?
Irq_Selected State
Current_Irq <= Current_Irq+1
Wait Until Interrupt Is Active

Fig. 3

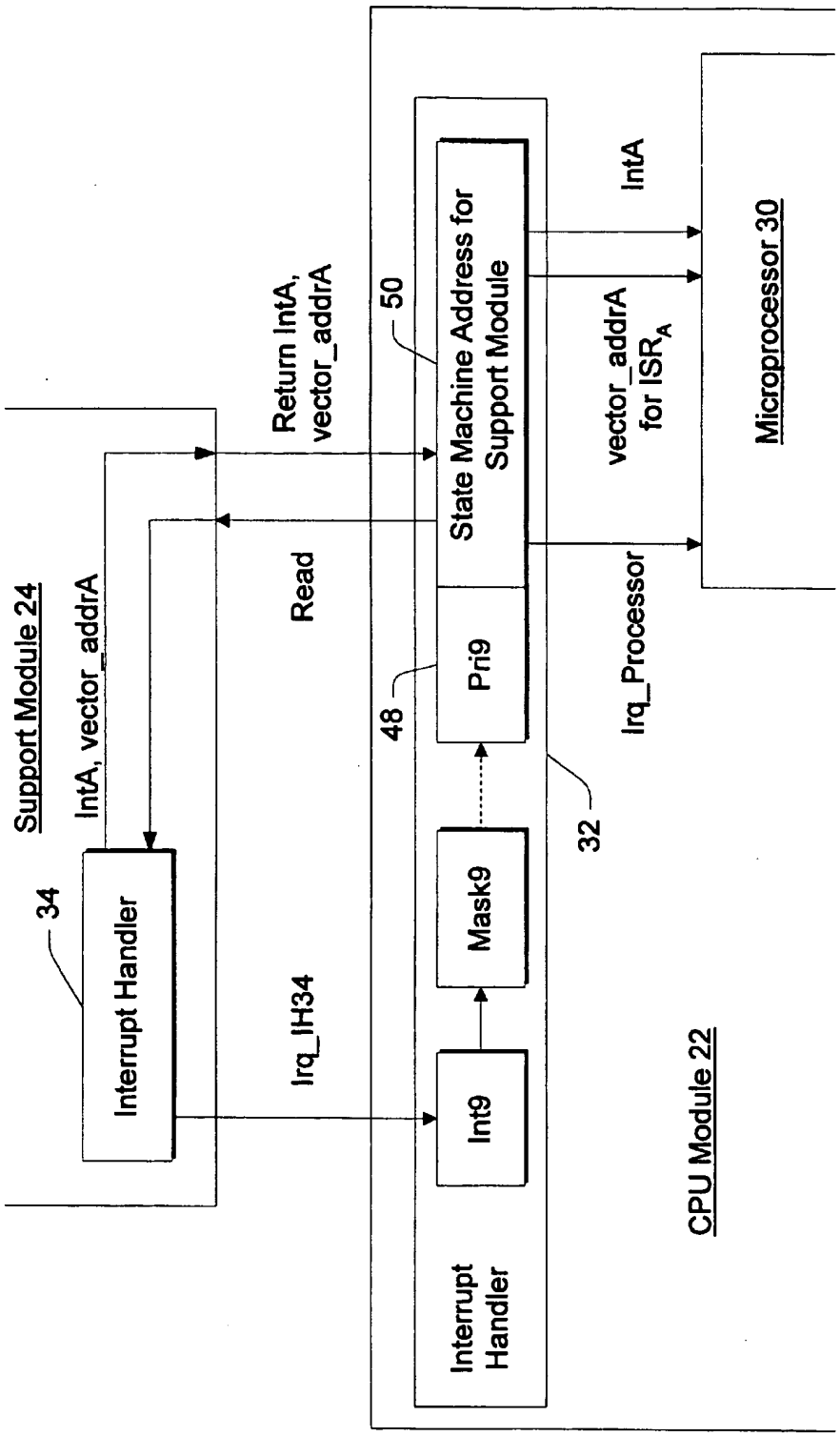

Support Module 24
34
Interrupt Handler
IntA, vector_addrA
Irq_IH34
Read
Return IntA, vector_addrA
50
State Machine Address for Support Module
48
Pri9
Mask9
Int9
Interrupt Handler
32
CPU Module 22
Irq_Processor
IntA
vector_addrA for ISR_A
Microprocessor 30

Fig. 4

INTERRUPT HANDLER WITH PRIORITIZED INTERRUPT VECTOR GENERATOR

TECHNICAL FIELD

This invention relates to interrupt handlers and methods for handling interrupts. More particularly, this invention relates to interrupt handlers with prioritized interrupt vector generators that enable configuration of interrupt type and priority among the interrupts.

BACKGROUND

An "interrupt" is a request-for-attention signal that is passed to a computer's CPU (central processing unit) from hardware or software sources in an attempt to gain the CPU's attention. Interrupts can occur for many reasons, ranging from normal to highly abnormal situations, including service requests from hardware, errors in processing, program attempts to do the impossible, and memory problems. A hardware interrupt is a request for service generated by hardware components, such as a keyboard, mouse, disk drive, I/O port, and microprocessor.

The interrupt causes the CPU to suspend its current operations, save the status of its work, and transfer control to a process for handling the interrupt. Interrupt handlers are commonly implemented in software and more particularly, in a hardware abstraction layer. The interrupt handler typically resides in the CPU at a known address. When an interrupt occurs, the CPU begins executing code at that location. The interrupt handler determines the cause of the interrupt and then services it by calling an appropriate set of instructions to be carried out.

The interrupt handler initiates different instructions for different types of interrupts. More specifically, each type of interrupt has an associated dedicated routine, known as an "interrupt service routine" or "ISR". When a CPU receives interrupt requests from more than one source, the interrupt handler invokes a hierarchy of permission levels, called "interrupt priorities", to determine which of the interrupts is handled first.

Conventional software-based interrupt handlers have a drawback in that the speed and performance is often unacceptable in real-time operating systems that are required to execute interrupts at very high speed and efficiency. The performance factor is further complicated by the desire to handle prioritized sets of interrupts from many diverse hardware platforms. Different hardware platforms often have different interrupts and dissimilar interrupt priorities. To be acceptable, an interrupt handler should provide real-time response for hardware interrupts and dynamic setting of interrupt priorities.

One prior art interrupt handler that addresses these issues is described in U.S. Pat. No. 5,594,905, entitled "Exception Handler and Method for Handling Interrupts", which issued Jan. 14, 1997 in the name of Amit Mital, and is assigned to Microsoft Corporation.

While this interrupt handler proved effective, the inventors sought to develop an even faster, hardware-based interrupt handler that minimizes software overhead to thereby improve performance.

SUMMARY

This invention concerns an interrupt handler implemented in hardware and external to a processor to handle interrupts destined for the processor. The interrupt handler has a programmable prioritized interrupt array with programmable registers that identify priority levels and handling processes for handling one or more interrupts. The interrupt handler also has an interrupt scanning state machine that scans the prioritized interrupt array following receipt of an interrupt to extract the priority level and handling process associated with the interrupt. The interrupt handler is designed to handle interrupts in significantly less time than software implementations, thereby making the handler favorable for real time systems.

According to one implementation, the prioritized interrupt array has three registers that coordinate data associated with the interrupts received by the interrupt handler. A mask register holds mask values for corresponding interrupts that indicate whether the interrupts are enabled. A priority register holds priority levels for the various interrupts, whereby two or more interrupts may be assigned the same priority level. An address register holds information for servicing the interrupts, such as addresses for interrupt service routines. The priority register and address register are programmable to enable a user to change the priority levels and servicing information for any given interrupt source.

The interrupt scanning state machine operates on the prioritized interrupt array using a three-state process that includes an idle state, a scanning state, and an interrupt selected state. The state machine remains in the idle state until the prioritized interrupt array receives one or more active interrupts that are indicated by the mask register as being enabled. When one or more active interrupts are received, the interrupt scanning state machine transitions to the scanning state to scan the priority register and identify which of the one or more active interrupts has the highest priority. When the interrupt with the highest priority is found, the interrupt scanning state machine transitions to the interrupt selected state and accesses the address register to output information for handling the active interrupt with the highest priority.

The interrupt handler is designed to be extendable. As an example, two or more handlers can be connected in a cascading arrangement where the output of one interrupt handler supplies its highest priority interrupt as one of the many interrupts to another interrupt handler.

BRIEF DESCRIPTION OF THE DRAWINGS

The same-reference numbers are used throughout the disclosure to reference like components and features.

FIG. **1** is a block diagram of a modular computer system with two interrupt handlers arranged in a cascaded relationship.

FIG. **2** is a block diagram of an interrupt handler.

FIG. **3** a state diagram implemented by the interrupt handler.

FIG. **4** is a flow diagram illustrating the process of handing an interrupt via the cascaded interrupt handlers.

CONCLUSION

Exemplary System

FIG. **1** shows a computer system **20** having a CPU (central processing unit) module **22** and at least one support module **24**. A bus structure **26** (e.g., PCI bus) interconnects the CPU module **22** and the support module **24**. The CPU module **22** has a microprocessor **30** and an interrupt handler **32** for handling interrupts or exceptions destined for the microprocessor. The interrupt handler **32** is constructed in hardware external to the microprocessor **30**.

The interrupt handler **32** receives one or more interrupts from different sources, evaluates the priorities of the interrupts, and derives servicing information for the highest priority interrupt. The interrupt handler **32** passes the servicing information and interrupt source number to the microprocessor **30** for servicing. The interrupt handler **32** implements a prioritized interrupt vector generator that enables the user to program the interrupt priorities and servicing information as desired.

The CPU module **22**, by itself, is generally representative of an embedded processor device. Aspects of this invention are directed to the interrupt handler and to embedded processor devices that incorporate the interrupt handler.

In addition, aspects of this invention concern the entire computer system **20**, whereby multiple interrupt handlers are arranged in a cascaded architecture. For instance, in the illustrated example, the support module **24** has an interrupt handler **34** that handles its own interrupts, evaluates the priorities of the interrupts, and passes its highest priority interrupt to the CPU-based interrupt handler **32**. In the illustrated example, the CPU-based interrupt handler **32** receives the interrupt from the remote interrupt handler **34** as one of many interrupts (e.g., Int**9**). The CPU-based interrupt handler **32** then evaluates the interrupt according to its own priorities. In this manner, the interrupt handler architecture is extensible to multiple interrupt handlers, thereby enabling the architecture to quickly handle large numbers of interrupts.

The computer system **20** is representative of many different computing systems, including general-purpose computers and dedicated computing devices. As one particular example, computer system **20** can be implemented as a modular vehicle computer as described in U.S. Pat. No. 5,794,164, entitled "Vehicle Computer System", which issued Aug. 11, 1998, in the names of Richard D. Beckert, Mark Moeller, and William Wong and is assigned to Microsoft Corporation. The vehicle computer has three modules—a computer module, a support module, and a faceplate module—two of which are shown. The computer module mounts in the vehicle dashboard or other location and runs an operating system to support vehicle-related applications and provide additional functionality typically afforded by a personal computer. The support module contains, for example, a storage drive (which also functions as an entertainment player), power supply, a communications bus, an AM/FM tuner, a DSP audio processor and a CODEC. The faceplate module (not shown) includes a display, a keypad and an IrDA interface. The drivers for all hardware components run in the host microprocessor **30**.

In the context of the vehicle computer, the remote interrupt handler **34** handles interrupts from the components supported by the support module **24**, such as the storage drive and DSP audio processor. The CPU-based interrupt handler **32** handles interrupts directed toward the computer processor on the CPU module **22**, such as interrupts from software and hardware sources as well as interrupts received from the remote interrupt handler **34** on the support module **24**.

Interrupt Handler

FIG. **2** shows the interrupt handler **32** in more detail. Interrupt handler **34** is constructed in a similar manner, and thus only one handler is described. The interrupt handler includes an N-dimensional prioritized interrupt array **40** constructed as hardware register. In the illustrated example, the interrupt array **40** has thirty-two entries (i.e., N=32). The array **40** receives N interrupts **42**, as represented by interrupts Int**0**–Int**31**, from various sources that provide interrupts to the microprocessor **30**.

The prioritized interrupt array **40** is partitioned into a mask register **44** and a priority/address register **46**. The registers each have N slots or entries that are associated within one another via the physical hardware structure. The mask register **44** is an N×A register having one entry for each associated interrupt source **42**. Each entry holds a mask value that indicates whether the associated interrupt is enabled. In one exemplary implementation, the mask register **42** is a 32×1-bit register (i.e., N=32 and A=1) whereby each interrupt source is allotted an an associated mask bit, represented by values mask**0**–mask**31**. The mask bit is active (i.e., a binary "1") if the interrupt is enabled and inactive (i.e., a binary "0") if the interrupt is disabled.

The priority/address register **46** is an N×(B+C) register having one entry for each associated interrupt source **42** and each corresponding entry in the mask register **44**. Each entry in the priority/address register **46** contains a B-bit priority field **48** that holds a priority level of the associated interrupt **42** and a C-bit address field **50** that holds servicing information to servicing the associated interrupt.

In the exemplary implementation, the priority field **48** holds a five bit value (i.e., B=5) ranging from Pri**0** (highest priority) to Pri**31** (lowest priority) for each of the interrupt sources. Different interrupt sources may be assigned the same priority level. For example, interrupts Int**3**, Int**4**, and Int**5** all have a priority Pri**4**. The address field **50** holds a 32-bit address (i.e., C=32) that specifies a start address for an interrupt service routine (ISR) that can be invoked to service the associated interrupt.

It is noted that the prioritized interrupt array **40** is expandable to have more or less than N slots and the bit sizes of the registers may vary depending upon the desired implementation. Furthermore, the priority/address register **46** may hold values other than ISR addresses. For instance, in the cascaded architecture described below, the address field **50** in the CPU-based interrupt handler **32** may hold an address received from the remote interrupt handler **34** on the support module **24** which is the highest priority interrupt from the remote interrupt handler **34**.

The interrupt handler **32** also has an interrupt scanning state machine **60** that scans the prioritized interrupt array **40** for all active interrupt sources. More particularly, when the interrupt handler **32** receives one or more of the interrupts that are specified by the mask register **44** as being enabled, an OR gate **62** produces an interrupt active signal "Irq_active" that starts the scanning state machine **60**. The scanning state machine scans the priority field **48** of the priority/address register **46** using mutiplexer **64** to discern what priority levels are associated with the active interrupts. Upon locating an active interrupt and its associated priority level, the state machine continues scanning for other active interrupts to see if there is still a higher interrupt pending. If two or more active interrupts have the same priority, the state machine will process the first interrupt source encountered in its scanning.

Upon determining the highest-priority active interrupt source to be serviced, the state machine **60** asserts three signals: (1) a "Irq_selected" signal indicating that a certain interrupt has been selected; (2) a "Highest_Irq" signal indicating that the selected interrupt is currently the highest priority; and (3) a "Irq_processor" signal that interrupts the processor **30**. When both the "Irq_selected" signal and the "Highest_Irq" signal are asserted, an AND gate **66** outputs the interrupt source number "Irq_Num". A multiplexer **68**

uses the source number to locate the corresponding vector address for the ISR from the address field **50** of the priority/address register **46**.

The interrupt handler **32** passes the interrupt source number and the vector address for the ISR to the processor. Upon receiving the interrupt signal "Irq_Processor", the processor simply jumps to the address specified in the vector supplied by the interrupt handler **32**. The output of scanning state machine **60** remains fixed until the interrupt service routine executed by the processor clears (resets) the interrupt source. At that time, the state machine will become active again, searching for a next highest priority interrupt to be serviced.

Operation

FIG. **3** is a state diagram showing a scanning process controlled by the scanning state machine **60**. When no interrupts are active, the state machine **60** sits in an Idle State **80**. When one or more enabled interrupts go active and the OR gate **62** outputs an interrupt active signal (i.e., Irq_active=1), the state machine enters a Scanning State **82**.

For discussion purposes, suppose that the interrupt handler **32** receives three interrupts: "Int6", "Int11", and "Int30". Further, suppose that corresponding mask values "mask6", "mask11", and "mask30" in mask register **44** indicate that all three interrupts are enabled.

In the Scanning State **82**, the state machine **60** scans the priority field **48** of the priority/address register **46** to determine which of the active interrupts has the highest programmed priority value. The state machine begins at the top of the array **40** and cycles through to the bottom to locate the interrupt source with the highest priority. The state machine initializes a "Current_Irq" count to zero, and sequentially steps through the array by incrementing by one the "Current_Irq".

In this example, the state machine first encounters the priority level "Pri11" for interrupt source "Int6". Thus far, this interrupt is initially considered to be the highest priority. The state machine **60** continues scanning until reaching the priority level "Pri3" of the next active interrupt "Int11". Priority level "Pri3" is higher than priority level "Pri11" and hence the state machine substitutes, the priority level "Pri3" as the highest priority.

Once again, the state machine **60** continues scanning until reaching the priority level "Pri3" of the third and final active interrupt "Int30". Notice that this interrupt source has the same priority level as the previous interrupt source. In this case, the scanning state machine **60** keeps the first-encountered interrupt source "Int11" as the highest priority interrupt. In this manner, the state machine **60** remembers the first-encountered interrupt source with the highest priority level.

After all sources have been scanned (i.e., Current_Irq=31), the state machine transitions to an Interrupt Selected State **84**. The state machine asserts the "Irq_Selected" signal and the "Highest_Irq" so that the AND gate **66** outputs the interrupt source number "Irq_Num" of the highest priority interrupt to be serviced. The multiplexor **68** locates the vector address in field **50** of the priority/address register **46** that corresponds to the highest priority interrupt and outputs that address. Here, the vector address "vector_addr11" is an address of the interrupt service routine corresponding to interrupt "Int11". The state machine **60** asserts the processor interrupt "Irq_Processor" and provides the interrupt source number "Int11" and corresponding vector address "vector_addr11" as outputs for the processor to read.

The state machine **60** stays in the Interrupt Selected State **84** until the interrupt "Int11" is inactive once again. The state machine then returns to the Idle State **80** and determines whether any enabled interrupts are active. In this example, interrupts "Int6" and "Int30" are still active and perhaps new interrupts have become active since the last scan. Therefore, the state machine **60** transitions immediately to the Scanning State **82** to locate the interrupt with the highest priority.

Cascaded Architecture

As noted above, multiple interrupt handlers may be arranged in a cascaded architecture where the output of one interrupt handler becomes an interrupt source to a next interrupt handler. For instance, in FIG. **1**, the output of the interrupt handler **34** on support module **24** becomes interrupt source "Int9" of the interrupt handler **32** on CPU module **22**.

FIG. **4** shows the cascaded architecture in more detail and describes its operation. Suppose that the interrupt handler **34** of support module **24**, through its own internal scanning operation like that described above with respect to FIG. **3**, determines that the interrupt source "IntA" is the highest priority interrupt currently pending. The remote interrupt handler **34** temporarily stores the interrupt source "IntA" and corresponding address "vector_addrA" for servicing the interrupt. The remote interrupt handler **34** then outputs an interrupt signal "Irq_IH34", which is received at the CPU-based interrupt handler **32** as interrupt source "Int9".

Assuming that the interrupt "Int9" is enabled (i.e., as indicated by mask**9** in interrupt handler **32**), the scanning state machine **60** of interrupt handler **32** scans the priority field **48** to locate the highest priority active interrupt. Assuming that only interrupt "Int9" is active or that it is the highest priority among active interrupts, the state machine **60** scans the address field **50** to locate a corresponding address.

In this case, the address field **50** holds an address for the state machine in the remote interrupt handler **34**. Upon finding this address, the CPU-based interrupt handler **32** starts a bus cycle on bus **26** to read the information currently being held at the remote interrupt handler **34** that corresponds to the interrupt source "IntA". The CPU-based interrupt handler **32** then outputs the interrupt source "IntA" and corresponding address "vector_addrA" to the processor **30** for servicing.

Programming Prioritized Interrupt Array

The prioritized interrupt array **40** is programmable to set the interrupt priority levels in field **48** and the corresponding information in field **50** for servicing the interrupts. To program the priority field **48**, an external chip select (CS) signal is applied to multiplexer **64** by the microprocessor **30** to locate the particular priority slot and a five-bit value is written to the slot using standard I/O or memory-write instructions. Similarly, to program the address field **50**, an external chip select (CS) signal is applied to multiplexer **68** by the microprocessor **30** to locate the particular address slot and a 32-bit value is written to the slot.

CONCLUSION

Although the invention has been described in language specific to structural features and/or methodological steps, it is to be understood that the invention defined in the appended claims is not necessarily limited to the specific features or steps described. Rather, the specific features and steps are disclosed as preferred forms of implementing the claimed invention.

What is claimed is:

1. An interrupt handler for handling interrupts, comprising:

a programmable prioritized interrupt array that stores priority levels for associated interrupts and information for handling the associated interrupt; and

a dynamic interrupt scanning state machine that scans the prioritized interrupt array upon receipt of one or more interrupts to locate the interrupt with a highest priority and identify the information for handling the interrupt with the highest priority.

2. An interrupt handler as recited in claim 1, wherein the information comprises addresses to interrupt service routines for servicing the associated interrupts.

3. A computer comprising:

a processor; and

an interrupt handler as recited in claim 1 to handle interrupts for the processor.

4. A vehicle computer for a vehicle comprising an interrupt handler as recited in claim 1.

5. A vehicle computer for a vehicle, comprising:

a CPU module with a processor and a first interrupt handler constructed as the interrupt handler recited in claim 1 to handle interrupts for the processor;

a bus connected to the CPU module; and

a support module connected to the bus, the support module having a second interrupt handler constructed as the interrupt handler recited in claim 1, the second interrupt handler supplying a highest priority interrupt derived at the support module as one of the interrupts handled by the first interrupt handler on the CPU module.

6. An interrupt handler for handling interrupts, comprising:

a programmable prioritized interrupt array that stores priority levels for associated interrupts and information for handling the associated interrupt; and

an interrupt scanning state machine that scans the prioritized interrupt array upon receipt of one or more interrupts to locate the interrupt with a highest priority and identify the information for handling the interrupt with the highest priority, wherein the information comprises at least one address to a remote interrupt handler.

7. An interrupt handler for handling interrupts, comprising:

a programmable prioritized interrupt array that stores priority levels for associated interrupts and information for handling the associated interrupt; and

an interrupt scanning state machine that scans the prioritized interrupt array upon receipt of one or more interrupts to locate the interrupt with a highest priority and identify the information for handling the interrupt with the highest priority, wherein the prioritized interrupt array may be programmed such that multiple different interrupts have identical priority levels.

8. An interrupt handler for handling interrupts, comprising:

a programmable prioritized interrupt array that stores priority levels for associated interrupts and information for handling the associated interrupt; and

an interrupt scanning state machine that scans the prioritized interrupt array upon receipt of one or more interrupts to locate the interrupt with a highest priority and identify the information for handling the interrupt with the highest priority, wherein the interrupt scanning state machine scans the prioritized interrupt array sequentially.

9. An interrupt handler for handling interrupts, comprising:

a programmable prioritized interrupt array that stores priority levels for associated interrupts and information for handling the associated interrupt; and

an interrupt scanning state machine that scans the prioritized interrupt array upon receipt of one or more interrupts to locate the interrupt with a highest priority and identify the information for handling the interrupt with the highest priority, wherein if the prioritized interrupt array locates an interrupt before scanning all of the prioritized interrupt array, the interrupt scanning state machine continues scanning until all of the prioritized interrupt array has been scanned to ensure that the interrupt with the highest priority is located.

10. An interrupt handler for handling interrupts, comprising:

a programmable prioritized interrupt array that stores priority levels for associated interrupts and information for handling the associated interrupt, wherein the prioritized interrupt array comprises:

a mask register to hold mask values for corresponding ones of the interrupts to indicate whether the interrupts are enabled; and

a priority/address register having a priority field to hold priority levels for the interrupts and an address field to hold addresses to interrupt service routines for servicing the associated interrupts; and

an interrupt scanning state machine that scans the prioritized interrupt array upon receipt of one or more interrupts to locate the interrupt with a highest priority and identify the information for handling the interrupt with the highest priority.

11. An interrupt handler for handling interrupts, comprising:

a programmable prioritized interrupt array that stores priority levels for associated interrupts and information for handling the associated interrupt; and

an interrupt scanning state machine that scans the prioritized interrupt array upon receipt of one or more interrupts to locate the interrupt with a highest priority and identify the information for handling the interrupt with the highest priority, wherein the interrupt scanning state machine is initially in an idle state until the prioritized interrupt array receives one or more interrupts; and whereupon receipt of the one or more interrupts, the interrupt scanning state machine transitions to a scanning state to scan the prioritized interrupt array and identify which of the one or more interrupts has a highest priority; and whereupon identifying the interrupt with the highest priority, the interrupt scanning state machine transitions to an interrupt selected state and accesses the prioritized interrupt array to output the information for handling the interrupt with the highest priority.

12. An interrupt handler for handling interrupts, comprising:

a programmable prioritized interrupt array for receiving one or more interrupts from multiple interrupt sources, the prioritized interrupt array having:

a mask register to hold mask values for corresponding interrupt sources to indicate whether the interrupts from the interrupt sources are enabled;

a priority register to hold priority levels for the interrupts received from the corresponding interrupt sources;

an address register to hold information for handling the interrupts received from the corresponding interrupt sources;

an interrupt scanning state machine coupled to the prioritized interrupt array, the interrupt scanning state machine remaining in an idle state until the prioritized interrupt array receives one or more active interrupts that are indicated by the mark register as being enabled;

whereeupon receipt of the one or more active interrupts, the interrupt scanning state machine transitions to a

scanning state to scan the priority register and identify which of the one or more active interrupts has a highest priority; and

whereupon identifying the active interrupt with the highest priority, the interrupt scanning state machine transitions to an interrupt selected state and accesses the address register to output information for handling the active interrupt with the highest priority.

13. An interrupt handler as recited in claim **12**, wherein the address register holds addresses to interrupt service routines for servicing the interrupts received from the corresponding interrupt sources.

14. An interrupt handler as recited in claim **12**, wherein the address register holds at least one address to a remote interrupt handler.

15. An interrupt handler as recited in claim **12**, wherein the priority register and the address register are programmable.

16. An interrupt handler as recited in claim **12**, wherein the interrupt scanning state machine scans the priority register sequentially.

17. A computer comprising:

a processor; and

an interrupt handler as recited in claim **12** to handle interrupts for the processor.

18. A vehicle computer for a vehicle comprising an interrupt handler as recited in claim **12**.

19. A computing device comprising:

a processor;

an interrupt handler implemented in hardware and external to the processor to handle interrupts destined for the processor, the interrupt handler having programmable registers that define (1) whether interrupts from particular interrupt sources are enabled, (2) priority levels of the interrupts, and (3) information for handling the interrupts; and

upon receiving one or more interrupts, the interrupt handler through a dynamic state machine identifies a highest priority from among the one or more interrupts and provides an interrupt source identity and handling information for the highest priority interrupt to the processor.

20. A computing device as recited in claim **19**, stores addresses to interrupt service routines for servicing the interrupts.

21. A computing device as recited in claim **19**, embodied as a vehicle computer designed to reside in a vehicle.

22. A computer system comprising:

a bus;

a computing device as recited in claim **19** coupled to the bus; and

a remote interrupt handler coupled to the bus to supply an external interrupt to the intedrrupt handler of the computing device.

23. A computing device comprising:

a processor;

an interrupt handler implemented in hardware and external to the processor to handle interrupts destined for the processor, the interrupt handler having programmable registers that define (1) whether interrupts from particular interrupt sources are enabled, (2) priority levels of the interrupts, and (3) information for handling the interrupts; and

upon receiving one or more interrupts, the interrupt handler identifies a highest priority from among the one or more interrupts and provides an interrupt source identity and handling information for the highest priority interrupt to the processor, wherein multiple different interrupts are assigned with identical priority levels.

24. A computing device comprising:

a processor;

an interrupt handler implemented in hardware and external to the processor to handle interrupts destined for the processor, the interrupt handler having programmable registers that define (1) whether interrupts from particular interrupt sources are enabled, (2) priority levels of the interrupts, and (3) information for handling the interrupts; and

upon receiving one or more interrupts, the interrupt handler identifies a highest priority from among the one or more interrupts and provides an interrupt source identity and handling information for the highest priority interrupt to the processor, wherein the interrupt handler comprises:

a programmable prioritized interrupt array that interrelates the programmable registers in a data structure; and

an interrupt scanning state machine that scans the prioritized interrupt array upon receipt of the one or more interrupts.

25. A computing device as recited in claim **24**, wherein the interrupt scanning state machine is initially in an idle state until the prioritized interrupt array receives the one or more interrupts; and whereupon receipt of the one or more interrupts, the interrupt scanning state machine transitions to a scanning state to scan the prioritized interrupt array and identify which of the one or more interrupts has a highest priority; and whereupon identifying the interrupt with the highest priority, the interrupt scanning state machine transitions to an interrupt selected state and accesses the prioritized interrupt array to output the information for handling the interrupt with the highest priority.

26. A computing device comprising:

a processor;

an interrupt handler implemented in hardware and external to the processor to handle interrupts destined for the processor, the interrupt handler having programmable registers that define (1) whether interrupts from particular interrupt sources are enabled, (2) priority levels of the interrupts, and (3) information for handling the interrupts; and

upon receiving one or more interrupts, the interrupt handler identifies a highest priority from among the one or more interrupts and provides an interrupt source identity and handling information for the highest priority interrupt to the processor, wherein one of the interrupt sources is a remote interrupt handler.

27. A method for handling interrupts comprising:

awaiting receipt of one or more interrupts;

upon receipt of the one or more interrupts, scanning using a dynamic state machine a prioritized interrupt array to determine which interrupt has a highest priority; and

upon identifying the interrupt with the highest priority, accessing the prioritized interrupt array to obtain information for handling the interrupt with the highest priority.

28. A method for handling interrupts comprising:

awaiting receipt of one or more interrupts;

upon receipt of the one or more interrupts, scanning a prioritized interrupt array to determine which interrupt has a highest priority;

upon identifying the interrupt with the highest priority, accessing the prioritized interrupt array to obtain infortnation for handling the interrupt with the highest priority; and

repeating the scanning and the accessing until all of the one or more interrupts have been handled.

* * * * *

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. : 6,499,078 B1
DATED : December 24, 2002
INVENTOR(S) : Beckert et al.

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

Column 4,
Line 12, delete the second occurrence of "an".

Column 8,
Line 66, replace "whereeupon" with -- whereupon --.

Column 9,
Line 52, replace "intedrrupt" with -- interrupt --.

Column 10,
Lines 61 and 62, replace "infortnation" with -- information --.

Signed and Sealed this

Eleventh Day of March, 2003

JAMES E. ROGAN
Director of the United States Patent and Trademark Office

US006633942B1

(12) **United States Patent**
Balasubramanian

(10) **Patent No.: US 6,633,942 B1**
(45) **Date of Patent: Oct. 14, 2003**

(54) **DISTRIBUTED REAL-TIME OPERATING SYSTEM PROVIDING INTEGRATED INTERRUPT MANAGEMENT**

(75) Inventor: **Sivaram Balasubramanian**, Mayfield Heights, OH (US)

(73) Assignee: **Rockwell Automation Technologies, Inc.**, Mayfield Heights, OH (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **09/408,670**

(22) Filed: **Sep. 30, 1999**

Related U.S. Application Data

(60) Provisional application No. 60/148,541, filed on Aug. 12, 1999.

(51) **Int. Cl.**[7] **G06F 13/26**

(52) **U.S. Cl.** **710/264**; 710/262; 709/100; 709/103

(58) **Field of Search** 710/260, 261, 710/262, 263, 264, 265, 266, 267, 268, 269; 709/100, 102, 103, 104, 105, 107, 108, 1; 713/502, 600, 601

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,012,409	A *	4/1991	Fletcher et al.	709/103
5,081,577	A *	1/1992	Hatle	710/7
5,297,275	A *	3/1994	Thayer	713/500
5,528,513	A *	6/1996	Vaitzblit et al.	709/103
5,542,076	A *	7/1996	Benson et al.	710/260
5,560,018	A *	9/1996	Macon et al.	710/260
5,560,019	A *	9/1996	Narad	710/260
5,640,563	A *	6/1997	Carmon	709/102
5,659,759	A *	8/1997	Yamada	710/265
5,768,599	A *	6/1998	Yokomizo	710/260
5,797,019	A *	8/1998	Levine et al.	710/262
5,937,199	A *	8/1999	Temple	710/262
6,085,215	A *	7/2000	Ramakrishnan et al.	709/102
6,092,095	A *	7/2000	Maytal	709/100
6,209,086	B1 *	3/2001	Chi et al.	712/244
6,430,594	B1 *	8/2002	Akiyama et al.	709/108

* cited by examiner

Primary Examiner—Sumati Lefkowitz
Assistant Examiner—X. Chung-Trans
(74) *Attorney, Agent, or Firm*—Quarles & Brady; Alexander M. Gerasimow; William R. Walbrun

(57) **ABSTRACT**

An interrupt handler is provided for a real-time control system that prevents interrupts which occur asynchronously with respect to control tasks from upsetting guarantees of timely execution of the control tasks. For interrupts associated with the communication of messages between portions of a control task over the distributed system, the interrupts are converted to proxy tasks that may be scheduled like any task in a multitasked-operated system. More generally, interrupts may be assigned to a predetermined interrupt window being a portion of the total processing bandwidth of the processor. In pre-allocating the processor bandwidth to the control tasks, this interrupt window may be subtracted out thereby guaranteeing adequate bandwidth for both interrupt processing and user tasks.

11 Claims, 6 Drawing Sheets

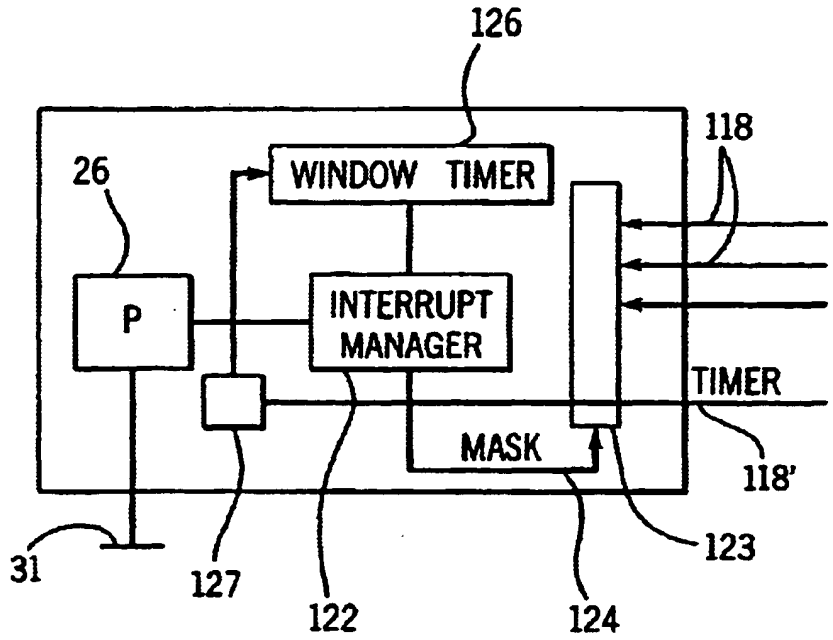


FIG. 1

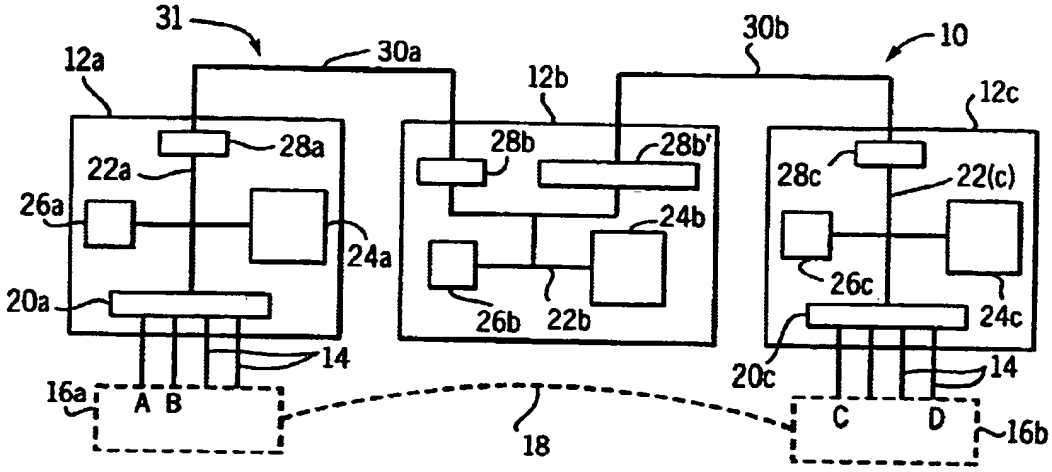
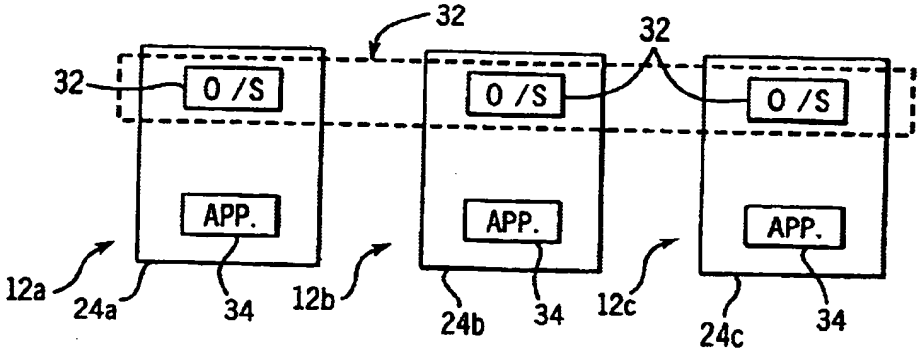

31
30a
30b
10
12a
12b
12c
28a
28b
28b'
28c
22a
22(c)
26a
24a
24b
26c
20a
26b
22b
24c
14
20c
16a
A
B
C
D
16b
18

FIG. 2


32
32
32
O /S
O /S
O /S
APP.
APP.
APP.
12a
24a
34
12b
24b
34
12c
24c
34

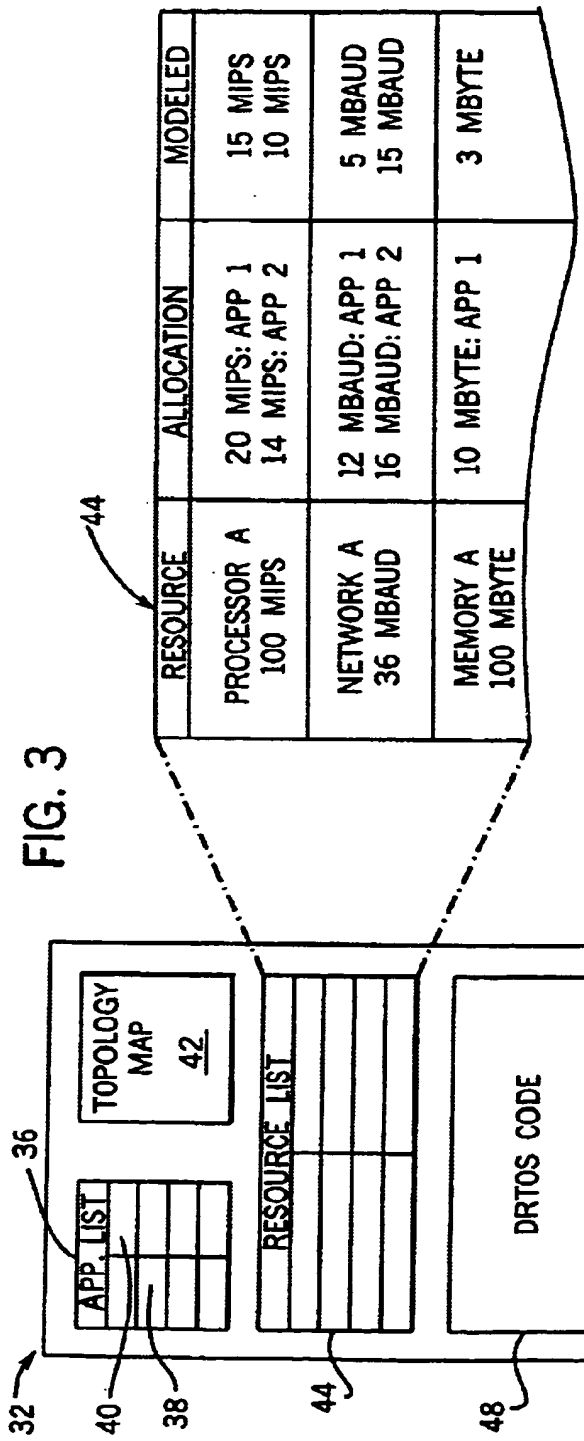

RESOURCE
ALLOCATION
MODELED
PROCESSOR A
100 MIPS
20 MIPS: APP 1
14 MIPS: APP 2
15 MIPS
10 MIPS
NETWORK A
36 MBAUD
12 MBAUD: APP 1
16 MBAUD: APP 2
5 MBAUD
15 MBAUD
MEMORY A
100 MBYTE
10 MBYTE: APP 1
3 MBYTE
44
TOPOLOGY MAP
42
APP. LIST
RESOURCE LIST
DRTOS CODE
36
32
40
38
44
48

FIG. 3

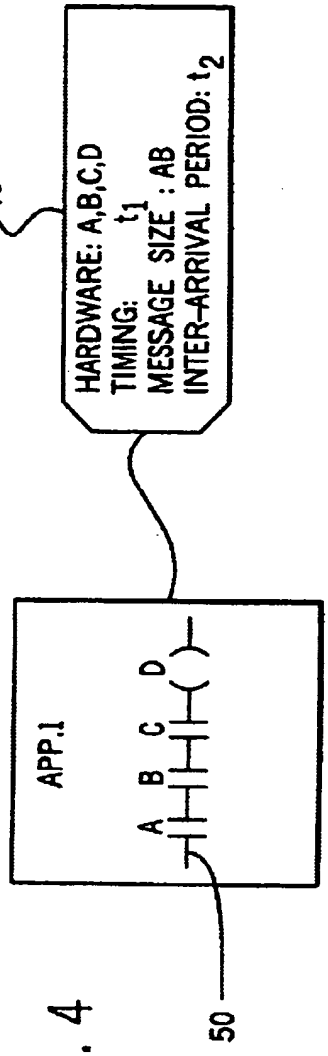

HARDWARE: A,B,C,D
TIMING: t_1
MESSAGE SIZE: AB
INTER-ARRIVAL PERIOD: t_2
40
APP.1
A
B
C
D
34
50

FIG. 4

FIG. 5

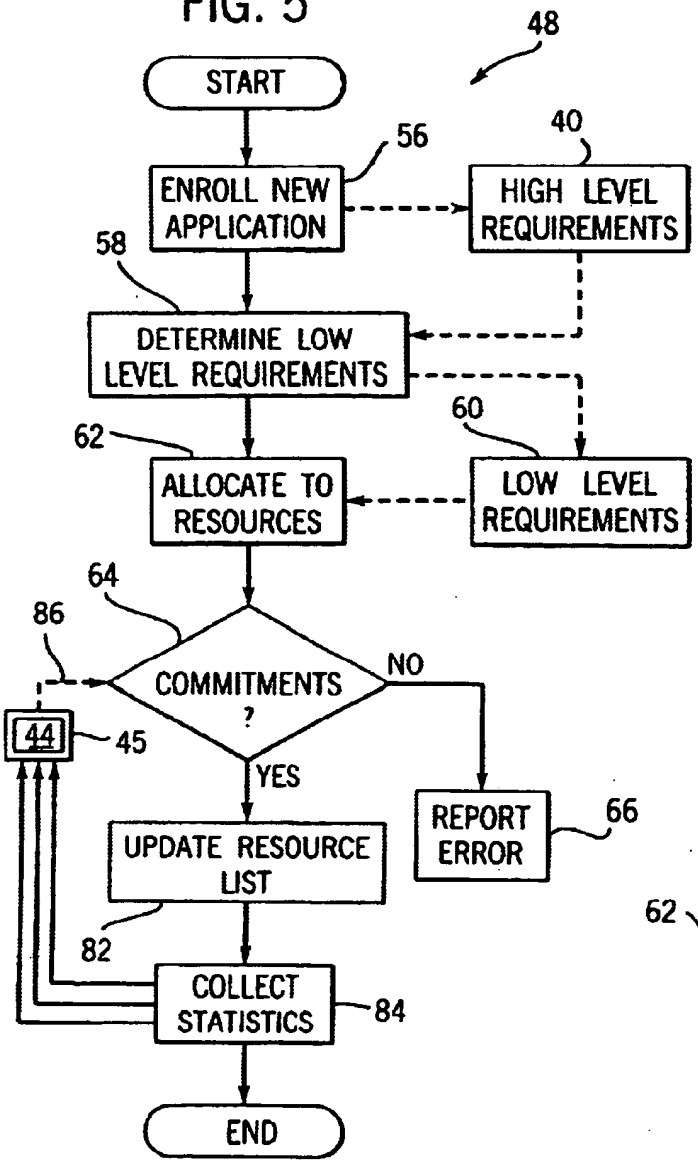


FIG. 7

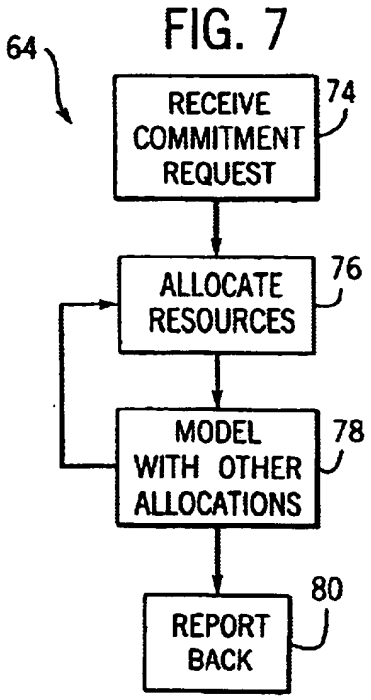
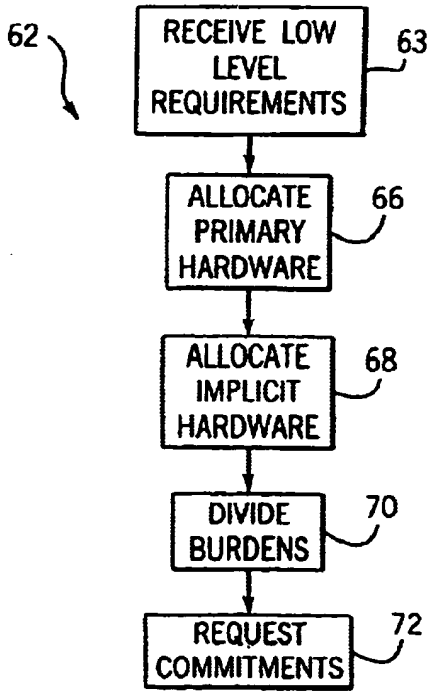


FIG. 6



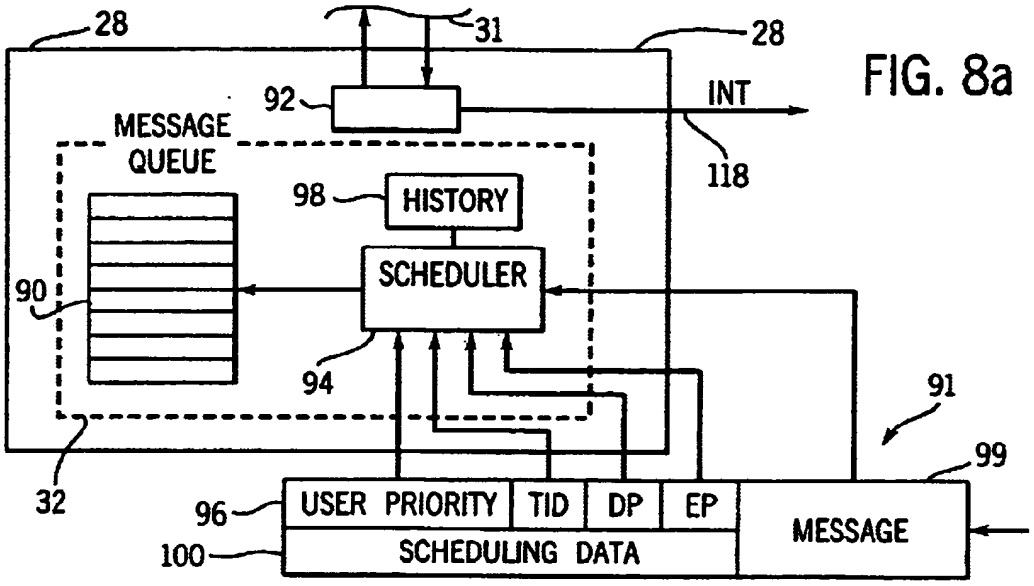

28
31
28
92
INT
MESSAGE QUEUE
118
98
HISTORY
SCHEDULER
90
94
91
99
32
96
USER PRIORITY
TID
DP
EP
MESSAGE
100
SCHEDULING DATA

FIG. 8a

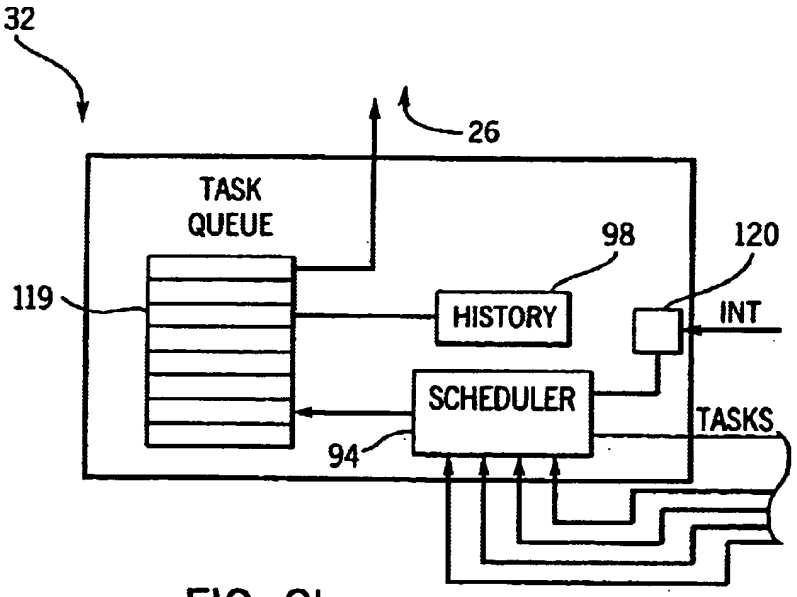

32
26
TASK QUEUE
98
120
119
HISTORY
INT
SCHEDULER
TASKS
94

FIG. 8b

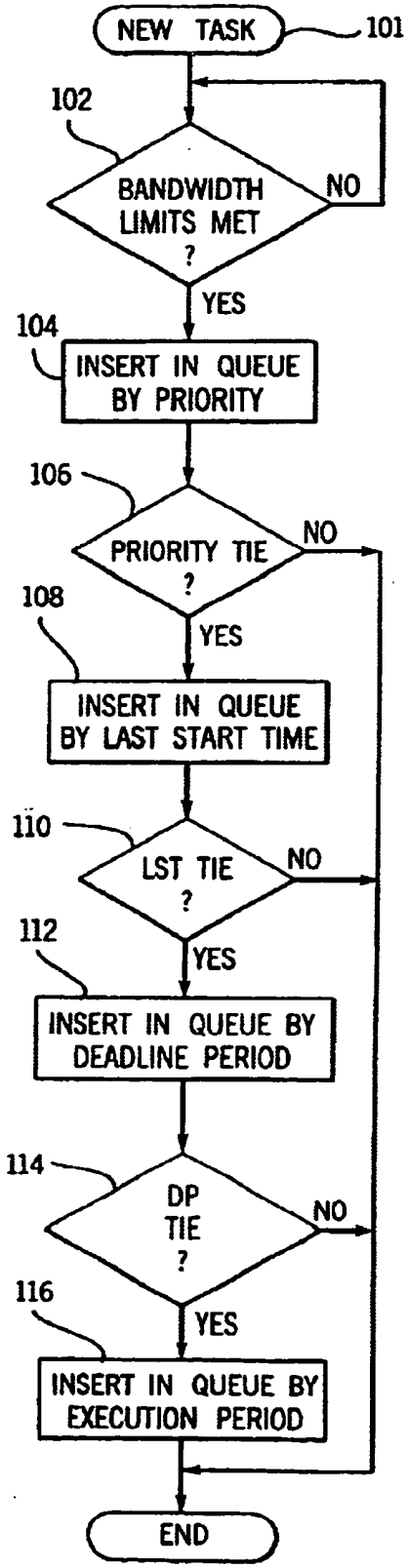
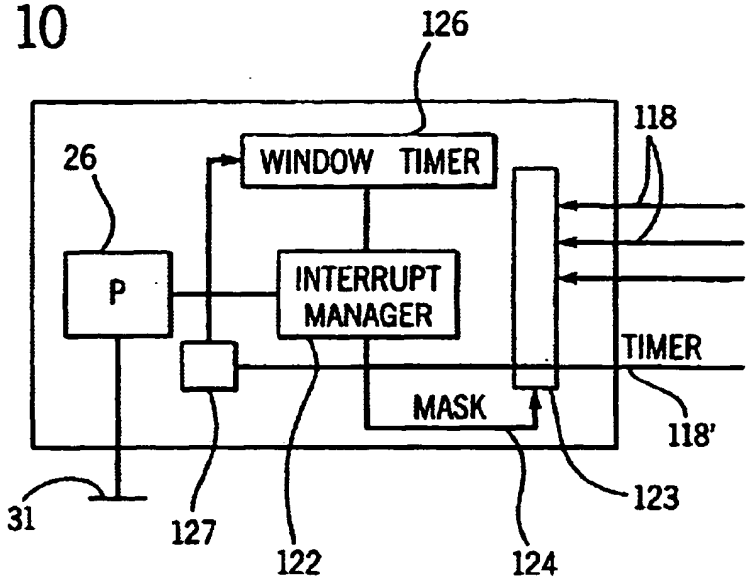

NEW TASK
101
102
BANDWIDTH LIMITS MET ?
NO
YES
104
INSERT IN QUEUE BY PRIORITY
106
PRIORITY TIE ?
NO
108
YES
INSERT IN QUEUE BY LAST START TIME
110
LST TIE ?
NO
112
YES
INSERT IN QUEUE BY DEADLINE PERIOD
114
DP TIE ?
NO
116
YES
INSERT IN QUEUE BY EXECUTION PERIOD
END

FIG. 9

FIG. 10


126
118
26
WINDOW TIMER
P
INTERRUPT MANAGER
TIMER
MASK
118'
123
31
127
122
124

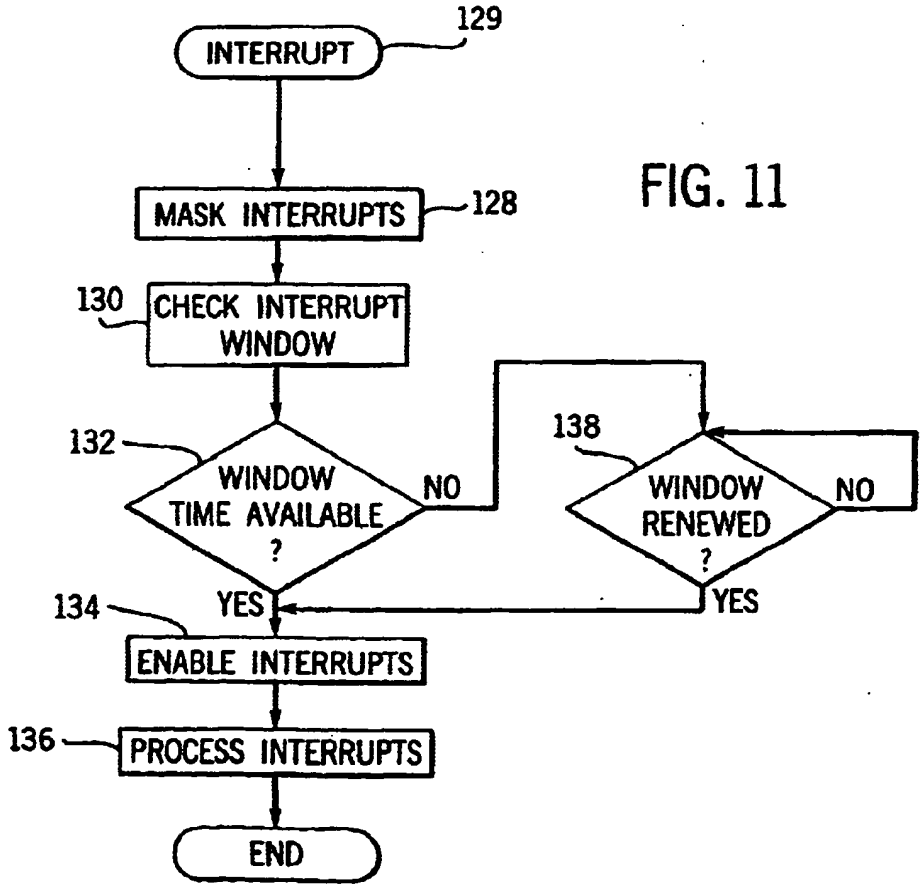

129
INTERRUPT
MASK INTERRUPTS
128
130
CHECK INTERRUPT WINDOW
132
WINDOW TIME AVAILABLE ?
NO
138
WINDOW RENEWED ?
NO
YES
YES
134
ENABLE INTERRUPTS
136
PROCESS INTERRUPTS
END

FIG. 11

DISTRIBUTED REAL-TIME OPERATING SYSTEM PROVIDING INTEGRATED INTERRUPT MANAGEMENT

CROSS-REFERENCE TO RELATED APPLICATIONS

This application claims the benefit of provisional application Ser. No. 60/148,541 filed Aug. 12, 1999.

STATEMENT REGARDING FEDERALLY SPONSORED RESEARCH OR DEVELOPMENT

BACKGROUND OF THE INVENTION

The present invention relates to industrial controllers for controlling industrial processes and equipment, and more generally to an operating system suitable for a distributed industrial control system having multiple processing nodes spatially separated about a factory or the like.

Industrial controllers are special purpose computers used for controlling industrial processes and manufacturing equipment. Under the direction of a stored control program the industrial controller examines a series of inputs reflecting the status of the controlled process and in response, adjusts a series of outputs controlling the industrial process. The inputs and outputs may be binary, that is on or off, or analog providing a value within a continuous range of values.

Centralized industrial controllers may receive electrical inputs from the controlled process through remote input/output (I/O) modules communicating with the industrial controller over a high-speed communication network. Outputs generated by the industrial controller are likewise transmitted over the network to the I/O circuits to be communicated to the controlled equipment. The network provides a simplified means of communicating signals over a factory environment without multiple wires and the attendant cost of installation.

Effective real-time control is provided by executing the control program repeatedly in high speed "scan" cycles. During each scan cycle each input is read and new outputs are computed. Together with the high-speed communications network, this ensures the response of the control program to changes in the inputs and its generation of outputs will be rapid. All information is dealt with centrally by a well-characterized processor and communicated over a known communication network to yield predictable delay times critical to deterministic control.

The centralized industrial controller architecture, however, is not readily scalable, and with foreseeably large and complex control problems, unacceptable delays will result from the large amount of data that must be communicated to a central location and from the demands placed on the centralized processor. For this reason, it may be desirable to adopt a distributed control architecture in which multiple processors perform portions of the control program at spatially separate locations about the factory. By distributing the control, multiple processors may be brought to bear on the control problem reducing the burden on any individual processor and the amount of input and output data that must be transmitted.

Unfortunately, the distributed control model is not as well characterized as far as guaranteeing performance as is required for real-time control. Delay in the execution of a portion of the control program by one processor can be fatal to successful real-time execution of the control program, and

because the demand for individual processor resources fluctuates, the potential for an unexpected overloading of a single processor is possible. This is particularly true when a number of different and independent application programs are executed on the distributed controller and where the application programs compete for the same set of physical hardware resources.

One problem in ensuring timely execution of tasks in a distributed environment arises in the processing of interrupts. Interrupts are electrical signals acting directly on the hardware of the processor to cause the processor to stop its current execution of a program and respond, typically, to an external device requiring immediate attention. Interrupts avoid inefficient polling by the processor of asynchronous signals and thus greatly improve the efficiency of some types of processing. Implicitly, interrupts normally have the highest priority with respect to response by the processor. A distributed control system with its attendant increased need for intercommunication among disparate components, may make extensive use of interrupts.

Unfortunately, because interrupts occur asynchronously to the execution of task on the processor, there exists the possibility that a large number of interrupts will occur within a small window of time thus preventing the timely execution of the task which is being interrupted. When the interrupts are caused by low priority tasks, the effects of this is a priority inversion where lower priority tasks displace higher priority tasks. This may lead to the failure of time critical tasks.

SUMMARY OF THE INVENTION

The present invention provides two methods of managing interrupts in the context of a real-time control system where task execution must proceed according to guaranteed completion times. For those interrupts associated with incoming messages and remote services provided by the operating system, the interrupts are embedded into a proxy task which is scheduled along with other tasks executed by a multitasking operating system. The proxy task may preempt the current task or may wait its turn depending on its priority.

All interrupts, are allocated to an interrupt window being a fixed percentage of time of the processor bandwidth. If interrupt window time is available, the interrupt is processed. Nested interrupting is allowed providing for a high degree of responsiveness of the control system. The interrupt window is subtracted from the of bandwidth of the processor that may be allocated in pre-allocation of bandwidth to application programs. Accordingly, guarantees of timely execution of programs having pre-allocated bandwidth may be ensured despite asynchronous interrupts such as may occur during run time.

Specifically, the present invention provides an interrupt manager for use with a processor forming part of a distributed control system. The interrupt manager includes interrupt reception circuitry receiving interrupt signals including a current interrupt. An interrupt window counter stores a value indicating the time remaining in a current window for the service of interrupts. The interrupt window counter is reset by a window timer at the expiration of each window period. A masking circuit masks current interrupts when the current interrupt would cause the value of the interrupt window counter to exceed the pre-allocated interrupt period.

Thus, it is one object of the invention to limit the servicing of interrupts to a finite period within each processing window. In this way, an arbitrary confluence of interrupts will

not upset the deterministic execution of control tasks that must adhere to deadlines. The interrupt manager may mask interrupts until the determination is made as to whether the current interrupt may be executed.

Thus, it is another object of the invention to allow the initial evaluation of an interrupt to proceed without further interruptions.

The interrupt manager may determine whether the current interrupt may be processed by adding an estimate of the time for processing the current value of the interrupt window counter. The estimation may be modified during actual execution of the interrupt.

Upon the determination that the current interrupt may be executed within the interrupt window, the interrupts are unmasked.

It is yet another object of the invention to permit nested interrupts such as provides for responsive operation of the interrupt process.

Thus, it is another object of the invention to provide for simple before-the-fact determination of whether an interrupt can proceed by allowing the use of a conservative estimate that is refined during run time.

It is yet another object of the invention to allow for the processing of subsequent nested interrupts by pre-estimating the amount of time required by each interrupt as it is received. In this way, nested interrupts may be accepted prior to an initial interrupt being completed.

Upon completion of the interrupt, the interrupt manager may add the estimate of the interrupt processing time and subtract the actual interrupt processing time from the value of the interrupt window counter.

Thus, it is another object of the invention to provide accurate accounting of actual interrupt time used while allowing pre-allocation of the interrupt time window.

The interrupt manager may cease masking the current interrupt upon resetting of the interrupt window counter by the timer.

It is, therefore, another object of the invention to allow stalled interrupts to nevertheless execute in order.

The interrupt manager may include a resource allocating operating system pre-allocating portions of the window period, excluding the predetermined interrupt window, to multiple tasks to be executed on the processor so as to guarantee timely execution of those tasks.

Thus, it is another object of the invention to allow pre-allocation of hardware resources to particular control tasks while guaranteeing interrupts will not usurp that allocation.

For the communication circuit in which the interrupts are related to incoming messages, the interrupt manager may include a task scheduler receiving tasks and arranging them in a queue according to priorities for execution by the processor. The communication circuit may receive messages having priorities to generate a communication interrupt. An interrupt reception circuit may receive the communication interrupts and the priorities and generate corresponding proxy tasks having the priority and enroll the proxy task on the task scheduler queue.

Thus, it is another object of the invention to provide a mechanism for processing interrupts for communication devices when messages form connections between tasks executed on spatially separate hardware making use of the same scheduling framework as the rest of the tasks thereby guaranteeing timely execution of the task as is necessary for real-time control.

The task scheduler may consider both the priority and time constraint value.

Thus, it is another object of the invention to provide for a mixed priority scheduling of interrupts through the use of a proxy task.

The foregoing and other objects and advantages of the invention will appear from the following description. In the description, reference is made to the accompanying drawings which form a part hereof and in which there is shown by way of illustration a preferred embodiment of the invention. Such embodiment does not necessarily represent the full scope of the invention, however, and reference must be made to the claims herein for interpreting the scope of the invention.

BRIEF DESCRIPTION OF THE SEVERAL VIEWS OF THE DRAWINGS

FIG. **1** is a simplified diagram of a distributed control system employing two end nodes and an intervening communication node and showing the processor, memory and communication resources for each node;

FIG. **2** is a block diagram showing the memory resources of each node of FIG. **1** as allocated to a distributed real-time operating system and different application programs;

FIG. **3** is an expanded block diagram of the distributed operating system of FIG. **2** such as includes an application list listing application programs to be executed by the distributed control system, a topology showing the topology of the connection of the hardware resources of the nodes of FIG. **1**, a resource list detailing the allocation of the hardware resources to the application program and the statistics of their use by each of the application programs, and the executable distributed real-time operating system code;

FIG. **4** is a pictorial representation of a simplified application program attached to its high-level requirements;

FIG. **5** is a flow chart of the operation of the distributed real-time operating system code of FIG. **3** showing steps upon accepting a new application program to determine the low-level hardware resource requirements and to seek commitments from those hardware resources for the requirements of the new application program;

FIG. **6** is a detailed version of the flow chart of FIG. **5** showing the process of allocating low-level requirements to hardware resources;

FIG. **7** is a block diagram detailing the step of the flow chart of FIG. **5** of responding to requests for commitment of hardware resources;

FIG. **8***a* is a detailed view of the communication circuit of FIG. 1 showing a messaging queue together with a scheduler and a history table as may be implemented via an operating system and showing a message received by the communication circuit over the bus of FIG. **1**;

FIGS. **8***b* is a figure similar to that of FIG. **8***a* showing the scheduler of FIG. **8***a* as implemented for multi-tasking of the processors of FIG. **1**;

FIG. **9** is a flow chart showing the steps of operation of enrolling the message of FIG. **8***a* or tasks of FIG. **8***b* into a queue;

FIG. **10** is a schematic representation of the interrupt handling system provided by the operating system and processor of FIGS. **1** and **2**; and

FIG. **11** is a flow chart showing the steps of operation of the interrupt handling system of FIG. **10**.

DETAILED DESCRIPTION OF THE INVENTION

Distributed Control System

Referring now to FIG. **1**, a distributed control system **10** includes multiple nodes **12***a*, **12***b* and **12***c* for executing a

control program comprised of multiple applications. Control end nodes **12***a* and **12***c* include signal lines **14** communicating between the end nodes **12***a* and **12***c* and a portion of a controlled process **16***a* and **16***b*. Controlled process portions **16***a* and **16***b* may communicate by a physical process flow or other paths of communication indicated generally as dotted line **18**.

In the present example, end node **12***a* may receive signals A and B from process **16***a*, and end node **12***c* may receive signal C from process **16***b* and provide as an output signal D to process **16***b* as part of a generalized control strategy.

End nodes **12***a* and **12***c* include interface circuitry **20***a* and **20***c*, respectively, communicating signals on signal lines **14** to internal buses **22***a* and **22***c*, respectively. The internal buses **22***a* and **22***c* may communicate with the hardware resources of memory **24***a*, processor **26***a* and communication card **28***a* (for end node **12***a*) and memory **24***c*, processor **26***c*, and network communication card **28***c* for end node **12***c*. Communication card **28***a* may communicate via network media **30***a* to a communication card **28***b* on node **12***b* which may communicate via internal bus **22***b* to memory **24***b* and processor **26***b* and to second network communication card **28***b* connected to media **30***b* which in turn communicates with communication card **28***c*.

Generally during operation of distributed control system application programs are allocated between memories **24***a*, **24***b* and **24***c* to be executed on the respective nodes **12***a*, **12***b* and **12***c* with communications as necessary over links **30***a* and **30***b*. In an example control task, it may be desired to produce signal D upon the logical conjunction of signals A, B and C. In such a control task, a program in memory **24***a* would monitor signals A and B and send a message indicating both were true, or in this example send a message indicating the state of signals A and B to node **12***c* via a path through communication **28***a*, **28***b*, **28***b*' and **28***c*.

A portion of the application program executed by processor **26***c* residing in memory **24***c* would detect the state of input C and compare it with the state of signals A and B in the received message to produce output signal D.

The proper execution of this simple distributed application program requires not only the allocation of the application program portions to the necessary nodes **12***a*, **12***b* and **12***c*, but prompt and reliable execution of those programs, the latter which requires the hardware resources of memory, processor, and communication networks **28***a*, **30***a*, **28***b*, **28***b*' **30***b* and **28***c*.

Referring now to FIG. **2** for this latter purpose, the distributed real-time operating system **32** of the present invention may be used such as may be centrally located in one node **12** or in keeping with the distributed nature of the control system distributed among the nodes **12***a*, **12***b* and **12***c*. In the latter case, the portions of the operating system **32** are stored in each of the memories **24***a*, **24***b* and **24***c* and intercommunicate to operate as a single system. In the preferred embodiment, a portion of the operating system **32** that provides a modeling of the hardware resources (as will be described) is located in the particular node **12***a*, **12***b* and **12***c* associated with those hardware resources. Thus, hardware resource of memory **24***a* in node **12***a* would be modeled by a portion of the operating system **32** held in memory **24***a*.

In addition to portions of the operating system **32**, memory **24***a*, **24***b* and **24***c* include various application programs **34** or portions of those application programs **34** as may be allocated to their respective nodes.

Integrated Resource Management

Referring now to FIG. **3**, the operating system **32** collectively provides a number of resources for ensuring proper operation of the distributed control system **10**. First, an application list **36** lists the application programs **34** that have been accepted for execution by the distributed control system **10**. Contained in the application list **36** are application identifiers **38** and high-level requirements **40** of the application programs as will be described below.

A hardware resource list **44** provides (as depicted in a first column) a comprehensive listing of each hardware resource of the distributed control system **10** indicating a quantitative measure of that resource. For example, for the principle hardware resources of processors **26**, networks **31** and memories **24**, quantitative measurements may be provided in terms of millions of instructions per second (MIPs) for processors **26**, numbers of megabytes for memories **24** and megabaud bandwidth for networks. While these are the principal hardware resources and their measures, it will be understood that other hardware resources may also be enrolled in this first column and other units of measures may be used. Generally, the measures are of "bandwidth", a term encompassing both an indication of the amount of data and the frequency of occurrence of the data that must be processed.

A second column of the hardware resource list **44** provides an allocation of the quantitative measure of the resource of a particular row to one or more application programs from the application list **36** identified by an application name. The application name may match the application identifier **38** of the application list **36** and the indicated allocation quantitative measure will typically be a portion of the quantitative measure of the first column.

A third column of the hardware resource list **44** provides an actual usage of the hardware resource by the application program as may be obtained by collecting statistics during running of the application programs. This measure will be statistical in nature and may be given in the units of the quantitative measure for the hardware resource provided in the first column.

The operating system **32** also includes a topology map **42** indicating the connection of the nodes **12***a*, **12***b* and **12***c* through the network **31** and the location of the hardware resources of the hardware resource list **44** in that topology.

Finally, the operating system also includes an operating system code **48** such as may read the application list **36**, the topology map **42**, and the hardware resource list **44** to ensure proper operation of the distributed control system **10**.

Referring now to FIG. **4**, each application program enrolled in the application list **36** is associated with high-level requirements **40** which will be used by the operating system code **48**. Generally, these high-level requirements **40** will be determined by the programmer based on the programmer's knowledge of the controlled process **16** and its requirements.

Thus, for the application described above with respect to FIG. **1**, the application program **34** may include a single ladder rung **50** (shown in FIG. **4**) providing for the logical ANDing of inputs A, B and C to produce an output D. The high-level requirements **40** would include hardware requirements for inputs and outputs A, B, C and D. The high-level requirements **40** may further include "completion-timing constraints" t_1 and indicating a constraint in execution time of the application program **34** needed for real-time control. Generally the completion-timing constraint is a maximum period of time that may elapse between occurrences of the last of inputs A, B and C to become logically true and the occurrence of the output signal D.

The high-level requirements **40** may also include a message size, in this case the size of a message AB which must

be sent over the network **31**, or this may be deduced automatically through use of the topology map **42** and an implicit allocation of the hardware.

Finally, the high-level requirements **40** include an "inter-arrival period" t_2 reflecting an assumption about the statistics of the controlled process **16***a* in demanding execution of the application program **34**. As a practical matter the inter-arrival period t_2 need be no greater than the scanning period of the input circuitry **20***a* and **20***c* which may be less than the possible bandwidth of the signals A, B and C but which will provide acceptable real-time response.

Referring now to FIG. **5**, the operating system code **48** ensures proper operation of the distributed control system **10** by checking that each new enrolled application program **34** will operate acceptably with the available hardware resources. Prior to any new application program **34** being added to the application list **36**, the operating system code **48** intervenes so as to ensure the necessary hardware resources are available and to ensure that time guarantees may be provided for execution of the application program.

At process block **56**, the operating system code **48** checks that the high-level requirements **40** have been identified for the application program. This identification may read a prepared file of the high-level requirements **40** or may solicit the programmer to input the necessary information about the high-level requirements **40** through a menu structure or the like, or may be semiautomatic involving a review of the application program **34** for its use of hardware resources and the like. As shown and described above with respect to FIG. **4**, principally four high-level requirements are anticipated that of hardware requirements, completion-timing constraints, message sizes, and the inter-arrival period. Other high-level requirements are possible including the need for remote system services, the type of priority of the application, etc.

Referring still to FIG. **5**, as indicated by process block **58**, the high-level requirements **40** are used to determine low-level requirements **60**. These low-level requirements may be generally "bandwidths" of particular hardware components such as are listed in the first column of the hardware resource list **44**. Generally, the low-level requirements will be a simple function of high-level requirements **40** and the objective characteristics of the application program **34**, the function depending on a priori knowledge about the hardware resource. For example, the amount of memory will be a function of the application program size whereas, the network bandwidth will be a function of the message size and the inter-arrival period t_2, and the processor bandwidth will be a function of the application program size and the inter-arrival period t_2 as will be evident to those of ordinary skill in the art. As will be seen, it is not necessary that the computation of the low-level requirements **60** be precise so long as it is a conservative estimate of low-level resources required.

The distinction between high-level requirements **40** and low-level requirements **60** is not fixed and in fact some high-level requirements, for example message size, may in fact be treated as low-level requirements as deduced from the topology map **42** as has been described.

Once the low-level requirements **60** have been determined at process block **62**, they are allocated to particular hardware elements distributed in the control system **10**. Referring also to FIG. **6**, the process block **62** includes sub-process block **63** where the low-level requirements abstracted at process block **58** are received. At process block **66**, end nodes **12***a* and **12***c* are identified based on their hardware links to inputs A, B and C and output D and a tentative allocation of the application program **34** to those nodes and an allocation of necessary processor bandwidth is made to these principal nodes **12***a* and **12***c*. Next at process block **68** with reference to the topology map **42**, the intermediary node **12***b* is identified together with the necessary network **31** and an allocation is made of network space based on message size and the inter-arrival period.

The burden of storing and executing the application program is then divided at process block **70** allocating to each of memories **24***a* and **24***c* (and possibly **12***b*), a certain amount of space for the application program **34** and to processors **26***a* and **26***c* (and possibly **26***b*) a certain amount of their bandwidth for the execution of the portions of the application program **34** based on the size of the application program **34** and the inter-arrival period t_2. Network cards **28***a*, **28***b*', **28***b* and **28***c* also have allocations to them based on the message size and the inter-arrival period t_2. Thus, generally the allocation of the application program **34** can include intermediate nodes **12***b* serving as bridges and routers where no computation will take place. For this reason, instances or portions of the operating system code **48** will also be associated with each of these implicit hardware resources.

There are a large number of different allocative mechanisms, however, in the preferred embodiment the application program is divided according to the nodes associated with its inputs per U.S. Pat. No. 5,896,289 to Struger issued Apr. 20, 1999 and entitled: "Output Weighted Partitioning Method for a Control Program in a Highly Distributed Control System" assigned to the same assignee as the present invention and hereby incorporated by reference..

During this allocation of the application program **34**, the completion-timing constraint t_1 for the application program **34** is divided among the primary hardware to which the application program **34** is allocated and the implicit hardware used to provide for communication between the possibly separated portions of the application program **34**. Thus, if the completion-timing constraint t_1 is nine milliseconds, a guaranty of time to produce an output after necessary input signals are received, then each node **12***a–c* will receive three microseconds of that allocation as a time obligation.

At process block **72**, a request for a commitment based on this allocation including the allocated time obligations and other low-level requirements **60** is made to portions of the operating system code **48** associated with each hardware element.

At decision block **64**, portions of the operating system code **48** associated with each node **12***a–c* and their hardware resources review the resources requested of them in processor, network, and memory bandwidth and the allocated time obligations and reports back as to whether those commitments may be made keeping within the allocated time obligation. If not, an error is reported at process block **66**. Generally, it is contemplated that code portions responsible for this determination will reside with the hardware resources which they allocate and thus may be provided with the necessary models of the hardware resources by the manufacturers.

This commitment process is generally represented by decision block **64** and is shown in more detail in FIG. **7** having a first process block **74** where a commitment request is received designating particular hardware resources and required bandwidths. At process block **76**, the portion of the operating system code **48** associated with the hardware element allocates the necessary hardware portion from hard-

ware resource list **44** possibly modeling it as shown in process block **78** with the other allocated resources of the resource list representing previously enrolled application programs **34** to see if the allocation can be made. In the case of the static resources such as memory, the allocation may simply be a checking of the hardware resource list **44** to see if sufficient memory is available. In dynamic resources such as the processors and the network, the modeling may determine whether scheduling may be performed such as will allow the necessary completion-timing constraints t_1 given the inter-arrival period t_2 of the particular application and other applications.

At the conclusion of the modeling and resource allocation including adjustments that may be necessary from the modeling at process block **80**, a report is made back to the other components of the operating system code **48**. If that report is that a commitment may be had for all hardware resources of the high-level requirements **40**, then the program proceeds to process block **82** instead of process block **66** representing the error condition as has been described.

At process block **82**, a master hardware resource list **44** is updated and the application program is enrolled in the application list **36** to run.

During execution of the application program **34** and as indicated by process block **84**, statistics are collected on its actual bandwidth usage for the particular hardware resources to which it is assigned. These are stored in the third column of the hardware resource list **44** shown in FIG. **3** and is shown in the block **45** associated with FIG. **5** and may be used to change the amount of allocation to particular application programs **34**, indicated by arrow **86**, so as to improve hardware resource utilization.

Scheduled Communication Queuing

Referring now to FIG. **8***a*, the communication card **28** will typically include a message queue **90** into which messages **91** are placed prior to being transmitted via a receiver/transmitter **92** onto the network **31**. A typical network queuing strategy of First-In-First-Out (FIFO) will introduce a variable delay in the transmission of messages caused by the amount of message traffic at any given time. Of particular importance, messages which require completion on a timely basis and which therefore have a high priority may nevertheless be queued behind lower level messages without time criticality. In such a queue **90**, priority and time constraints are disregarded, therefore even if ample network bandwidth is available and suitable priority attached to messages **91** associated with control tasks, the completion timing constraints t_1 cannot be guaranteed.

To overcome this limitation, the communication card **28** of the present invention includes a queue-level scheduler **94** which may receive messages **91** and place them in the queue **90** in a desired order of execution that is independent of the arrival time of the message **91**. The scheduler **94** receives the messages **91** and places them in the queue **90** and includes memory **98** holding a history of execution of messages identified to their tasks as will be described below. Generally the blocks of the queue **90**, the scheduler **94** and the memory **98** are realized as a portion of the operating system **32**, however, they may alternatively be realized as an application specific integrated circuit (ASIC) as will be understood in the art.

Each message **91** associated with an application program for which a time constraint exists (guaranteed tasks) to be transmitted by the communication card **28** will contain conventional message data **99** such as may include substantive data of the message and the routing information of the message necessary for transmission on the network **31**. In addition, the message **91** will also include scheduling data **100** which may be physically attached to the message data **99** or associated with the message data **99** by the operating system **32**.

The scheduling data **100** includes a user-assigned priority **96** generally indicating a high priority for messages associated with time critical tasks. The priority **96** is taken from the priority of the application program **34** of which the message **91** form a part and is determined prior to application program based on the importance of its control task as determined by the user.

The scheduling data **100** may also include a execution period (EP) indicating the length of time anticipated to be necessary to execute the message for transmission on the network **31** and a deadline period (DP) being in this case the portion of the completion timing constraint t_1 allocated to the particular communication card **28** for transmission of the message **91**. The scheduling data **100** also includes a task identification (TID) identifying the particular message **91** to an application program **34** so that the high level requirements of the application program **34**, imputed to the message **91** as will be described, may be determined from the application list **30** described above, and so that the resources and bandwidths allocated to the application program and its portion held in resource list **44** can be accessed by the communication card **28** and the scheduler **94**.

The scheduling data **100** may be attached by the operating system **32** and in the simplest case is derived from data entered by the control system programmer. The execution period after entry, may be tracked by the operating system during run-time and modified based on that tracking to provide for accurate estimations of the execution period over time.

Upon arrival of a message at the communication card **28**, the scheduling data **100** and the message data **99** are provided to the scheduler **94**. The scheduler **94** notes the arrival time based on a system clock (not shown) and calculates a LATEST STARTING TIME for the message (LST) as equal to a deadline time minus the execution period. The deadline time is calculated as the message arrival time plus the deadline period provided in the message.

Referring now to FIG. **9**, arrival of the message at the communication card **28** is indicated generally at process block **101** and is represented generally as a task, reflecting the fact that the same scheduling system may be used for other than messages as will be described below.

Following process block **101** is decision block **102** which determines whether the bandwidth limits for the task have been violated. The determination of bandwidth limits at block **102** considers, for example, the inter-arrival period t_2 for the messages **91**. A message **91** will not be scheduled for transmission until the specified inter-arrival period t_2 expires for the previous transmission of the message **91**. The expiration time of the inter-arrival period t_2 is stored in the history memory **98** identified to the TID of the message. This ensures that all guarantees for message execution can be honored. More generally for a task other than a message, the bandwidth limits may include processor time or memory allocations.

If at process block **102**, there is no remaining allocation of network bandwidth for the particular task and the task is guaranteed, it is not executed until the bandwidth again becomes available.

At succeeding block **104**, if the bandwidth limits have not been violated, the message is placed in the queue **90** according to its user priority **96**. Thus, high priority messages always precedes low priority messages in the queue **90**. The locking out of low priority messages is prevented by the fact that the high priority messages must have guaranteed bandwidths and a portion of the total bandwidth for each resource, the communication card **28**, for example, is reserved for low priority tasks.

At decision block **106**, it is determined whether there is a priority tie, meaning that there is another message **91** in the queue **90** with the same priority as the current message **91**. If not, the current message **91** is enrolled in the queue **90** and its position need not be recalculated although its relative location in the queue **90** may change as additional messages are enrolled.

If at decision block **106** there is a priority tie, the scheduler **94** proceeds to process block **108** and the messages with identical priorities are examined to determine which has the earliest LATEST STARTING TIME. The LATEST STARTING TIME as described above is an absolute time value indicating when the task must be started. As described above the LATEST STARTING TIME need only be computed once and therefore doesn't cause unbounded numbers of context switches. The current message is placed in order among the message of a similar priority according to the LATEST STARTING TIME with earliest LATEST STARTING TIME first.

If at succeeding process block **110**, there is no tie between the LATEST STARTING TIMES, then the enrollment process is complete. Otherwise, the scheduler **94** proceeds to process block **112** and the messages are examined to determine their deadline periods DP as contained in the scheduling data **100**. A task with a shorter deadline period is accorded the higher priority in the queue **90** on the rationale that shorter deadline periods indicate relative urgency.

At succeeding process block **114** if there remains a tie according to the above criteria between messages **91**, then at process block **116**, the tie is broken according to the execution period, EP, of the messages **91**. Here the rationale is that in the case of transient overload, executing the task with the shortest execution period will ensure execution of the greatest number of tasks.

A system clock with sufficient resolution will prevent a tie beyond this point by ensuring that the LATEST STARTING TIMES are highly distinct.

These steps of determining priority may be simplified by concatenating the relevant scheduling data **100** into a single binary value of sufficient length. The user priority forms the most significant bits of this value and execution period the least significant bits. This binary value may then be examined to place the messages (or tasks) in the queue **90**.

As each message **91** rises to the top of the queue **90** for transmission, its LATEST STARTING TIME is examined to see if it has been satisfied. Failure of the task to execute in a timely fashion may be readily determined and reported.

Mixed Priority Multi-Tasking

As mentioned, the scheduling system used for the communication card **28** described above is equally applicable to scheduling other resources within the distributed operating system, for example, the processors **26**. Referring to FIG. **8***b*, each processor **26** may be associated with a task queue **119** being substantially identical to the message queue **90** except that each slot in the task queue **119** may represent a particular bandwidth or time slice of processor usage. In this way, enrolling a task in the task list not only determines the order of execution but allocates a particular amount of processor resources to that task. New tasks are received again by a scheduler **94** retaining a history of the execution of the task according to task identification (TID) in memory **98** and enrolling the tasks in one of the time slots of the task queue **119** to be forwarded to the processor **26** at the appropriate moment. The tasks include similar tasks scheduling data as shown in FIG. **8***a* but need not include a message data **99** and may rely on the TID to identify the task implicitly without the need for copying the task into a message for actual transmission.

Referring to FIG. **9**, the operation of the scheduler **94** as with the case of messages above, only allocates to the task the number of time slots in the queue **90** as was reserved in its bandwidth allocation in the resource list **44**. In this way, it can be assured that time guarantees may be enforced by the operating system.

Interrupt Management

As is understood in the art, interrupts normally act directly on the processor **26** to cause the processor **26** to interrupt execution of a current task and to jump to an interrupt subroutine and execute that subroutine to completion before returning to the task that was interrupted. The interrupt process involves changing the value of the program counter to the interrupt vector and saving the necessary stack and registers to allow resumption of the interrupt routine upon completion. Typically interrupt signals may be masked by software instructions such as may be utilized by the operating system in realizing the mechanism to be described now.

Referring now to FIGS. **8***a* and **8***b*, a similar problem to that described above, of lower priority messages blocking the execution of higher priority messages in the message queue **90**, may occur with interrupts. For example, a system may be executing a time critical user task when a low priority interrupt, such as that which may occur upon receipt of low priority messages, may occur. Since interrupts are serviced implicitly at a high priority level, the interrupt effects a priority inversion with the high priority task waiting for the low priority task. If many interrupts occur, the high priority tasks may miss its time guarantee.

This problem may be solved in two ways. Referring to FIG. **8***a* upon a receipt of a message from network **31** and particularly those associated with remote operating system services, an interrupt **118** may be generated and passed to a task generator **120** shown in FIG. **8***b*. The task generator **120** which receives the interrupt generates a proxy task forwarded to the scheduler **94**. The proxy task assumes the scheduling data **100** of the message causing the interrupt and is subject to the same mixed processing as the tasks described above via the scheduler **94**. Depending on its priority and other scheduling data **100**, the proxy task may preempt the current task or might wait its turn. This procedure guarantees deterministic packet reception without affecting tasks on the receiving node adversely.

Referring now to FIG. **10** in an alternate form of interrupt management, interrupts **118** from general sources such as communication ports and other external devices are received by an interrupt manager **122** prior to invoking the interrupt hardware on the processor **26**. One exception to this is the timer interrupt **118'** which provides a regular timer "click" for the system clock which, as described above, is used by the scheduler **94**. The interrupt manager **122** provides a masking line **124** to a interrupt storage register **123**, the

masking line allowing the interrupt manager **122** to mask or block other interrupts (while storing them for later acceptance) and communicates with an interrupt window counter **126** which is periodically reset by a clock **127**.

Generally, the interrupt manager **122**, its masking line **124**, the interrupt storage register **123**, the interrupt window counter **126** and the window timer are realized by the operating system **32** but as will be understood in the art may also be implemented by discrete circuitry such as an application specific integrated circuit (ASIC).

Referring to FIG. **11**, the interrupt manager **122** operates so that upon the occurrence of an interrupt as indicated by process block **129**, all further interrupts are masked as indicated by process block **128**. The interrupt window counter **126** is then checked to see if a pre-allocated window of time for processing interrupts (the interrupt window) has been exhausted. The interrupt window is a percentage of processing time or bandwidth of processor **26** reserved for interrupts and its exact value will depend on a number of variables such as processor speed, the number of external interrupts expected and how long interrupts take to be serviced and is selected by the control system programmer. In the allocation of processor resources described above, the interrupt period is subtracted out prior to allocation to the various application programs. The interrupt window counter **126** is reset to its full value on a periodic basis by the clock **127** so as to implement the appropriate percentage of processing time.

At process block **130**, after the masking of the interrupts at process block **128**, the interrupt window counter **126** is checked to see if the amount of remaining interrupt window is sufficient to allow processing of the current interrupt based on its expected execution period. The execution periods may be entered by the control system programmer and keyed to the interrupt type and number. If sufficient time remains in the interrupt window, the execution period is subtracted from the interrupt window and, as determined by decision block **132**, then the interrupt manager **122** proceeds to process block **134**. At process block **134** the interrupts **118** are re-enabled via masking line **124** and at process block **136**, the current interrupt is processed. The time taken to process the interrupts monitored and at its conclusion the interrupt window is corrected by adding the estimated execution period and subtracting the actual monitored execution period.

By re-enabling the interrupts at process block **134**, nested interrupts may occur which may also be subject to the processing described with respect to process block **129**. Nested interrupting is possible because prior to each nested interrupt, the estimated execution period will be cleared against the interrupt window.

If at decision block **132**, there is inadequate time left in the interrupt window, then the interrupt manager **122** proceeds to decision block **138** where it remains until the interrupt window is reset by the clock **127**. At that time, process blocks **134** and **136** may be executed.

As mentioned, the interrupt window is subtracted from the bandwidth of the processor **26** that may be allocated to user tasks and therefore the allocation of bandwidth for guaranteeing the execution of user tasks is done under the assumption that the full interrupt window will be used by interrupts taking the highest priority. In this way, interrupts may be executed within the interrupt window without affecting guarantees for task execution.

The above description has been that of a preferred embodiment of the present invention. It will occur to those that practice the art that many modifications may be made without departing from the spirit and scope of the invention. In order to apprise the public of the various embodiments that may fall within the scope of the invention, the following claims are made.

I claim:

1. An interrupt manager for use with a processor in a distributed control system, the interrupt manager comprising:

(a) interrupt reception circuitry receiving interrupt signals including a current interrupt;

(b) an interrupt window counter storing an interrupt window indicating time available for processing of interrupts;

(c) a timer refreshing the interrupt window counter at expiration of a window period; and

(e) a masking circuit masking a current interrupt when the interrupt window counter indicates that the processing of the current interrupt would exceed the interrupt window in the current window period.

2. The interrupt manager of claim **1** wherein the interrupt reception circuitry allows masking of interrupts and wherein the interrupt manager upon receiving the interrupt signal masks further interrupts until it is determined that the processing of the current interrupt would not exceed the interrupt window in the current window period.

3. The interrupt manager of claim **1** wherein the interrupt manager provides an estimate of the time needed for processing the current interrupt and uses this estimate to change the value of the interrupt window when the interrupt window indicates that the processing of the current interrupt would not exceed the interrupt window.

4. The interrupt manager of claim **1** wherein the interrupt manager ceases masking the current interrupt upon the resetting of the interrupt window counter by the timer.

5. The interrupt manger of claim **1** including further a resource allocating operating system pre-allocating portions of the window period, excluding the maximum interrupt window to multiple tasks to be executed on the processor so as to guarantee timely execution of those tasks.

6. The interrupt manager of claim **1** wherein the interrupt manager determines whether the processing of the current interrupt would exceed the interrupt window of the current window period by subtracting an estimate of the interrupt processing time from the interrupt window.

7. The interrupt manager of claim **6** wherein the interrupt manager further monitors the execution of the current interrupt and modifies the estimate of the interrupt processing time according the monitoring;

whereby the interrupt processing time is more accurately determined.

8. The interrupt manager of claim **6** wherein upon completion of the current interrupt, the interrupt manager adds the estimate of the interrupt processing time to the interrupt window and subtracts an actual interrupt processing time from the value of the interrupt window;

whereby later estimates of the interrupt window are more accurate.

9. An interrupt manager for use with a processor in a distributed control system, the interrupt manager comprising:

(a) a task scheduler receiving tasks and arranging tasks in a queue according to priorities for execution by the processor;

(b) a communication circuit receiving messages having a priority to generate a communications interrupt; and

(c) an interrupt reception circuit communicating with the task scheduler and the communication circuit to receive communication interrupts and to generate corresponding proxy tasks sent to the task scheduler, the proxy task when executed by the processor causing the communication interrupt to be processed as a task, the proxy tasks having the priority of the message associated with their communication interrupt.

10. The interrupt manager of claim **9** wherein the message priority includes a user assigned priority and a time constraint value and wherein the interrupt reception circuit generates a proxy task having both the user assigned priority and the time constraint value task scheduler.

11. The interrupt manager of claim **10** wherein the task scheduler schedules the task in an order that depends on both the user assigned priority and the time constraint value.

* * * * *